Modelling and Analyses of Embedded Systems Design

Brekling, Aske Wiid

Publication date:
2010

Document Version
Publisher's PDF, also known as Version of record

Modelling and Analyses of Embedded Systems Design

Aske Brekling

Kongens Lyngby 2010
IMM-PHD-2010-236

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-PHD: ISSN 0909-3192

Summary

Designing embedded systems that are guaranteed to act correctly in any given situation is an extremely hard task. Modern embedded systems consist of applications that are typically executed on multi-core execution platforms. Analyzing timing aspects of such multi-core platforms is particularly difficult, as they may exhibit counter-intuitive behavior - so-called timing anomalies - where e.g. faster execution time of an individual part of a system may cause increased execution time for the system as a whole. In the work described here, we focus on the timely aspects and show examples of systems containing such timing anomalies.

The ARTS framework is an abstract simulation framework for embedded systems that is used where an application meets an execution platform. The application is described by a task graph and the execution platform through its processing elements and interconnects. A mapping of tasks onto the processing elements concludes a system described in ARTS. In this simulation framework, designers can - early in the design process - conduct design space exploration and analyze different configurations and setups of systems in terms of their timely behavior and resource usage. As analysis is based on simulation, only indications of the system behavior can be identified, but no guarantees can be given.

In the work presented in this dissertation, we aim at capturing this scenario: an application, an execution platform and the mapping, in a formal model. On the basis of this formal model, we conduct verification of real-time constraints and guarantee that the system acts correctly in terms of meeting all its timely requirements.

We present the MoVES languages: a language with which embedded systems can be specified at a stage in the development process where an application is identified and should be mapped to an execution platform (potentially multi-core).

We give a formal model for MoVES that captures and gives semantics to the elements of specifications in the MoVES language. We show that even for seemingly simple systems, the complexity of verifying real-time constraints can be overwhelming - but we give an upper limit to the size of the search-space that needs examining. Furthermore, the formal model exposes important scheduling situations that become central in establishing timed-automata models that can be used for analysis of MoVES specifications effectively.

Finally we present the MoVES tool, which can conduct automatic verification of interesting properties of MoVES specifications. In several examples, we use the MoVES tool to conduct analysis that identifies timing anomalies. We also conduct design space exploration in an example using the MoVES tool. And we show that it can be used for analysis of systems that, in size, resemble industrially-interesting systems.

We find that semantically-based verification is a promising approach for assisting developers of embedded systems. We provide examples of system verifications that, in size and complexity, point in the direction of industrially-interesting systems.

Resumé

At designe indlejrede systemer der kan garanteres at virke korrekt i enhver situation, er vanskeligt. Moderne indlejrede systemer er anvendelser, som typisk afvikles på platforme med flere processorer - såkaldte kerner - og at analysere tidslige aspekter af sådanne multi-kerne platforme er især svært, da de kan udvise kontraintuitiv adfærd - såkaldte tidslige anomalier - hvor for eksempel hurtigere eksekveringstid for en enkelt del af systemet kan resultere i en langsommere eksekveringstid for hele systemet. Med dette arbejde fokuserer vi på tidslige aspekter og vi viser eksempler på systemer med sådanne tidslige anomalier.

Framework'et ARTS er et abstrakt simuleringsframework for indlejrede systemer, der benyttes til analyse, hvor anvendelser møder eksekveringsplatforme. Anvendelser er beskrevet ved task-grafer og eksekveringsplatforme i form af processeringselementer og deres forbindelser. Efter mapning af de individuelle tasks til processeringselementer er et fuldt system beskrevet i ARTS. I dette simuleringsframework kan designere tidligt i design processen afprøve forskellige design af systemer i form af tidslige aspekter og aspekter vedrørende brug af resourcer. Eftersom analysen er baseret på simulering, kan den kun bruges som indikation på, hvordan systemet opfører sig, men der kan på ingen måde gives garantier i forhold til systemets tidslige krav.

I denne afhandling går vi efter at indfange netop dette scenarie; anvendelser, eksekveringsplatforme og mapning i en formel model. På basis af denne model kan verifikation af realtidskrav af tasks eksekveret på kerner udføres, og der kan gives garantier om alle systemets tidslige krav.

Vi præsenterer sproget MoVES, et sprog med hvilket indlejrede systemer kan specificeres på et tidspunkt i designprocessen, hvor anvendelser er identificeret,

og disse skal mappes på eksekveringsplatforme potentielt med flere kerner.

Vi giver en formel model, der indfanger og giver semantik til specifikationer beskrevet i sproget MoVES. Vi viser, at det at verificere realtidskrav for selv tilsyneladende simple systemer kan have stor kompleksitet, men vi giver en øvre grænse for den del af søgerummet, der skal undersøges. Den formelle model viser derudover, hvordan vigtige skeduleringssituationer er centrale når man skal udvikle tidsautomatimplementeringer der kan bruges til at analysere MoVES specificationer effektivt.

Sidst men ikke mindst præsenterer vi værktøjet MoVES, som kan udføre automatisk verifikation af interessante egenskaber på baggrund af MoVES specifikationer. Vi benytter værktøjet på en række eksempler, hvor tidslige anomalier identificeres. Vi giver også eksempler på, hvorledes man kan afprøve forskellige design af systemer (design space exploration) med værktøjet, og vi viser, at det kan benyttes til analyse af systemer, som i størrelse kunne have industriel interesse.

Vi fastslår, at semantisk baseret verifikation er en lovende mulighed for udviklere af indlejrede systemer, og vi viser eksempler på systemverifikationer, som i størrelse og kompleksitet peger i retning af for industrien interessante systemer.

Preface

This dissertation was prepared at DTU Informatics, the Technical University of Denmark, in partial fulfillment of the requirements for acquiring the degree of Doctor of Philosophy.

The dissertation deals with analysis and verification of embedded systems. The main focus is to develop languages, methods and tools that assist developers in early stages of the process of designing embedded systems.

The dissertation is self contained and relies on the work done in a number of research papers written during the period 2006–2010.

The PhD project has been funded by MoDES (Danish Research Council 2106-05-002).

The collaboration on the MoDES and DaNES projects has inspired most of this work, with input and examples from the academic partners: Technical University of Denmark, Aalborg University and University of Southern Denmark, as well as industrial partners: Hardi International A/S, Skov A/S, Danfoss A/S, Reactive Systems inc, CSI Center for Software Innovation, PAJ Systemteknik, ICEpower, Novo Nordisk, Terma and Prevas.

Lyngby, 2010

Aske Brekling

Acknowledgements

First and foremost I would like to thank my main supervisor, Michael R. Hansen, for his support and encouragement over the years. He has been available for discussion and comments at all times, and has been a source for both inspiration and guidance in all aspects of the project. He has been the most perfect supervisor anyone could ever ask for. Without him, this work would not have been possible at all.

Also thanks to my other supervisor, Jan Madsen. He has provided great insight in the area of embedded systems and the current state of the art research. Jan has an incredible way of looking at issues from different angles. This has made it possible to think out-of-the-box and come up with very interesting solutions and ideas.

Part of the work was carried out while visiting Virginia Tech University and Aalborg University. I would like to thank Patrick Schaumont for inviting me and for welcoming me at Virginia Tech. My stay gave great insight into how hardware descriptions can be done at higher abstraction levels, and how the Gezel language can be used as part of development. I also got to see how he uses Gezel in the courses given on Introduction to Hardware/Software Co-design. Also thanks to Kim G. Larsen for inviting me and giving me a chance to take part of the inspiring research community at Aalborg University. While visiting, I got the chance to meet developers of the Uppaal system and see presentations of some of the newest developments. I also got a chance to discuss issues of modelling with timed-automata with some of the researchers.

Thanks to all of the fellow PhD students in the Embedded Systems Engineering

section at DTU Informatics. Together we have created a great atmosphere that supports both the social sides and generates room for the exchange of opinions that can lead to great synergies in our work. I especially would like to thank Per Larsen, Peter V. B. Sørensen, Stavros Passas and Pascal Schleuniger for sharing office space with me. We have had some great times, and inspiring talks.

Thanks to everyone I have met throughout the past years. People that I have discussed and turned an amazing amount of interesting stones with. Kristian S. Knudsen and Jens Ellebæk were during their Master's work a great driving factor for the development of MoVES. My fellow university students... special thanks to Anders Høeg Dohn, Pavel Kozin and Søren G. Christensen for keeping me sane, and to all the great people I met and had great moments with at the Marktoberdorf summer school 2008.

Finally, the greatest thanks goes to my family; my mother and father, my siblings and their families - you are the strong base that every inspiring thought and all development in my thoughts comes from. My daughter Adia, who can make me continue that extra mile, and who makes me realize why I do the things I do. The love of my life, my wife Jeanifer! Always supportive, understanding and helpful. You make it possible for me to succeed.

Contents

Chapter 1

Introduction

Embedded systems are computer systems that are integrated into any kind of system - physical, mechanical, etc, and which are not easily accessible. They are characterized as being able to conduct one or more dedicated tasks for that specific system. There are increasing uses of embedded systems within many industries: avionics, automotive, medical equipment and consumer electronics, just to name a few.

Let us take an example from the automotive industry. A modern car can have embedded systems consisting of about 100 electronic control units. These units can execute tasks in anything from the climate control system to entertainment systems such as radio, navigation, video, etc, to the anti-lock braking system. This allows for capabilities that are much more advanced than could otherwise be included in the car. In this way, the user can enjoy these added features. It also offers more safety because it can use complex computation to adjust the behavior of the car in critical situations.

Within the area of consumer electronics, embedded systems practically are the products. Handheld gadgets such as multimedia players, GPS navigation equipment and smart phones are systems that can be viewed as a complete embedded system with just a simple interface to the user in terms of keypad, screen, loudspeakers, headphones, etc. For example, when handheld CD-players were popular, they comprised mechanical components that turned the disk, read it,

etc., together with the electronics. But today's MP3 players have no mechanical components. They are basically embedded systems that contain processing elements, storage, interconnections and simple user interfaces.

Dangers of faulty embedded systems

Some embedded systems are extremely safety critical. Consider, for example, the airbag safety system in a car. In the case of an accident, if the airbag does not deploy (or deploys milliseconds too early or too late), it can have fatal consequences. Within avionics, practically every component is considered safety critical; even a small error here can be life threatening. In 1996, Ariane 5's (Figure 1.1) first test flight, resulting in self destruction 37 seconds after launch, is one of the best-known faulty embedded systems. A data conversion from a 64-bit floating point value to a 16-bit signed integer value was the cause of the rocket's total destruction.

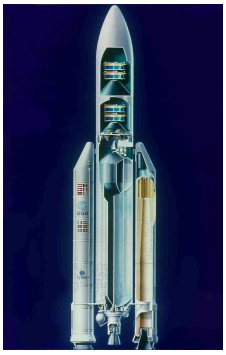


Figure 1.1: Ariane 5

The area of medical electronics is obviously safety critical as well: The case of Therac-25 (Figure 1.2) resulted in six incidents between 1985 and 1987, where patients were given massive radiation overdose caused by a faulty embedded system in the radiation accelerator.

The Mars Pathfinder mission (see the Mars Pathfinder on Figure 1.3) is an example where timing properties and resource usage resulted in system failure. After a few days on Mars, the space craft began experiencing total resets, resulting in loss of data. It turned out to be a case of priority inversion in a concurrent execution context. If the problem had not been fixed, the whole mission would have been a total failure. Fortunately, the onboard software could be modified and the mission resumed with complete success.

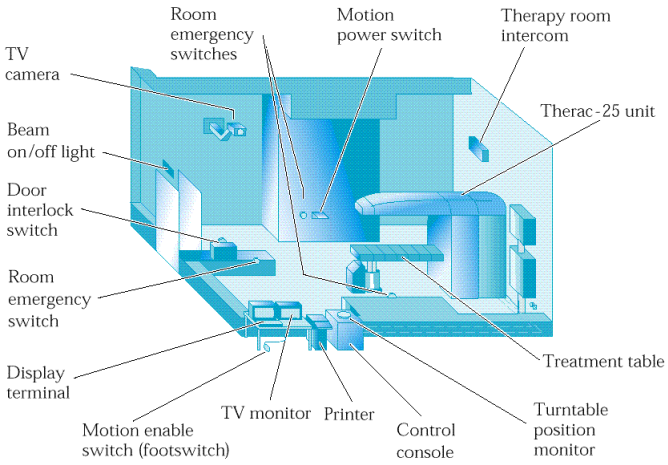


Figure 1.2: Typical Therac-25 facility

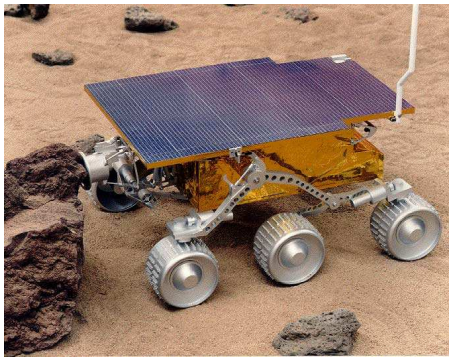


Figure 1.3: Mars Pathfinder

For many embedded systems, especially within consumer electronics, flawed systems can have a detrimental effect on the image of the producing company. If a manufacturer of cell phones releases a model that has a tendency to freeze up and needs to be reset often, users quickly communicate this to each other, and many people would come to prefer a different manufacturer for their next product. In the case of severe manufacturing defects, the product may need to be recalled. This both tarnishes the brand's reputation and is an extremely lengthy and expensive process. A flawed system like this can result in a decrease in market shares.

Whether the consequences of erroneous systems are dangerous or financially disastrous, a way to avoid such errors is much needed. The earlier in the design process these errors can be detected the better. However, these systems are growing in numbers; they are being used in a wider range of industries and are becoming more and more complex. It therefore becomes more difficult to detect system flaws. This is why it is crucial to use a systematic approach of analysis and constructive development. This approach should allow designers to make decisions when some details of the system are still undetermined, in order to detect possible flaws early in the design process.

1.1 Different Approaches for Analysis of Embedded Systems

Model-based development of embedded systems [53] has become an important discipline. This development start with abstract specifications of the system in terms of functional- and non-functional requirements. Functional requirements specify what the system is supposed to do, whereas non-functional requirements specify issues such as timing and other resource constraints on the system. Through a number of refinement steps, the development may end at a stage where the tasks of the systems have been identified together with their interdependencies, and a platform for executing the system is identified together with a mapping of the tasks onto the platform.

In order to avoid development of faulty systems, analyses of designs are needed at every stage of the development process. A key goal for such analyses is to determine whether the system is able to run without faults. If a system is defined as a number of tasks being executed and each task (or the system as a whole) having deadlines, this analysis will determine if any deadline of the system is missed, i.e. the system is not schedulable.
Analyses of systems that execute several tasks or processes have been studied extensively the last decades [7, 43]. This section will attempt to provide an overview of different approaches starting with the very general topic addressed by classic schedulability theory and ending with approaches specifically aimed at embedded systems.

1.1.1 Classic schedulability analysis

Liu and Layland [42] analytically studied schedulability of systems consisting of multiple tasks sharing processing elements as early as 1973. Classic schedula-

bility theory originated in scheduling of multiple tasks on a single processor but has developed into the area of distributed systems.

1.1.1.1 Single processor

In [42], Liu and Layland studied the problem of multiple task scheduling on a single processor. The results in this paper include an upper bound to processor utilization for an optimum fixed-priority scheduler (i.e. a rate-monotonic scheduler). It is optimal in the sense that if a system is schedulable using any fixed-priority scheduler, it is schedulable using rate-monotonic scheduling. This upper bound can be calculated as follows:

$$\sum_{i=1}^{m}(C_i/T_i) \leq m(2^{1/m} - 1)$$

where m is the number of tasks in the system, C_i and T_i are the execution time and the period of task i, respectively.

Also, it is shown that full processor utilization can be achieved by dynamically assigning priorities based on their current deadlines (i.e. earliest-deadline-first scheduling), provided that

$$\sum_{i=1}^{m}(C_i/T_i) \leq 1$$

Much of the later research in the area of schedulability analysis is based on these results. See e.g. [13, 14] for an overview of some of the later research results.

In [64, 63], Tindell extends these results to a more general approach for analyzing fixed priority hard real-time tasks. This approach captures aspects such as static priority preemptive systems, arbitrary deadlines and release jitter. This analysis uses a window approach to find worst-case response times. The window-based approach is based on the general concept of a busy period. The busy period is the maximum time that a task has to wait for higher prioritized tasks before it can execute. This busy period is divided up into a sequence of windows that correspond to each invocation of tasks with higher priority. When tasks of a system have different periods and if some tasks' deadlines are greater than their periods, a number of windows need to be examined to find the worst-case response time. In general the system's worst case response time could correspond to the response time in any one of the windows.

In a simple case where deadlines for all tasks are smaller than or equal to their periods, the worst-case response time r_i for task i can be computed by the following recursive function:

$$r_i = C_i + \sum_{j \in hp(i)} \lceil \frac{r_i}{T_j} \rceil$$

where C_i is the execution time for task i, $hp(i)$ is the set of tasks with higher priority than task i and T_j is the period of task j.

For cases where some deadlines for tasks are greater than their periods, this equation becomes more complex as it has to take into account the response time in any one of the windows. Schedulability analysis in this context is to calculate the worst-case execution times for all tasks, and to ensure that they are all less than their deadlines.

1.1.1.2 Distributed systems

When analyzing distributed systems, much more complex issues arise compared with analyzing single processor systems. In distributed systems, several tasks can execute concurrently, however, the issue of communication between processors must be addressed. If dependencies among tasks executing on different processors are taken into account, some timing anomalies can arise, i.e. local worst-case behavior does not give global worst-case behavior (when the whole system is addressed).

In [65], Tindell and Clark extend the analysis approach associated with static priority preemptive based scheduling with a so-called "holistic" approach. It is holistic in the sense that it takes the results from single processor scheduling theory and combines it with communication analysis to get a holistic view of the system. It addresses schedulability of distributed hard real-time systems. Specifically analysis of tasks with arbitrary deadlines, message parsing and shared data is derived. In all communication a simple TDMA protocol is assumed. The same window-based analysis technique as described for single processor analysis is used to find worst-case response times of distributed task sets. This approach introduces more complexity to the equations from the single-processor case. In the holistic approach, message parsing is included as tasks that need their response times calculated. Also, the time it takes to deliver a message at a receiving processor is taken into account.

The approaches presented in [63, 64, 65] were refined and extended by Palencia et al. in [24, 25]. This was done by completing the proof of validity of the schedulability analysis technique as well as including offset information into the analysis. Thereby, an increase of the maximum schedulable utilization was achieved. In [29], González Harbour et al. describe the MAST [16] toolset. It contains several schedulability analysis tools capable of analyzing single processor and distributed systems. The tools are based on different scheduling analysis techniques, including the aforementioned approaches by Tindell and Clark as well as Palencia et al.

In [58], Pop et al. propose an optimization strategy for bus accesses in distributed embedded systems based on classic schedulability analysis, in particular the approach by Tindel and Clark [65]. The communication model is based on a time-triggered protocol and analysis for communication delays is presented.

The classic scheduling theory addresses issues regarding shared resources and blocking, however, the concept of dependencies among tasks is not directly addressed. Therefore, the aforementioned timing anomalies cannot be analyzed through classic scheduling theory. This model is coarse in the sense that for concrete systems with data dependencies it is non-trivial to give approximations of the communication overhead.

1.1.2 Event stream analysis

The classic schedulability theory is not easily adapted to include analysis of heterogeneous systems. Furthermore, the complexity of the equations in the underlying analysis increases dramatically with size of systems. In [61], Thiele et al. propose a real-time calculus for schedulability analysis. In this analysis, a link between three areas is established: Max-Plus Linear System Theory [18], Network Calculus [17] and real-time scheduling [42]. In particular, the notions of request curves as a model for task behavior and delivery curves modelling hardware components' service of tasks are introduced. This provides a compositional approach, where each task executing on a hardware component is analyzed individually and given request- and service curves, which are then propagated through the distributed system. By the use and operations on these curves, an over-approximation is used. Thus, the results of these analyses do not provide exact results, and some results may be overly pessimistic. This work has resulted in the Real-Time Calculus (RTC) toolbox [67], which is a free Matlab toolbox for system-level performance analysis of distributed real-time and embedded systems.

In [32], Henia et al. present the SymTA/S approach. This work is based on the same compositional idea as in the work of Thiele et al., but instead of using generic request- and delivery curves and therefore having to introduce new complex stream representations, the notion of standard event models is presented (e.g. periodic-, sporadic-, periodic with jitter event models). These event models are described by sets of parameters. An example of an event model is a periodic with jitter event model, which has parameters for the period (i.e. time between periodic occurrence) and the jitter (i.e. the interval in which the exact occurrence takes place). Also, in the SymTA/S approach an over-approximation is used in the analysis, like with the Real-Time Calculus.

1.1.3 Timed-automata analysis

In [5], Alur and Dill propose timed (finite) automata to model the behavior of real-time systems over time. Timed-automata theory is shown to be adequate for automatic verification of real-time requirements of finite-state systems. Real-time properties of such systems can be expressed as reachability problems of their timed automata models, e.g. in timed temporal logics. Several model checkers such as Kronos [68, 20] and Uppaal [40, 8, 66] are available for automatic verification of such requirements.

There have been several examples of using timed-automata theory to model scheduling problems and in analyzing embedded systems: Abededdaïm and Maler show in [2] how the classic job-shop scheduling problem can be modelled as timed automata. In [1], this strategy is extended and includes problems with uncertainty in task durations. In [3], Altisen and Tripakis propose an implementation methodology for transformation of a timed automaton into a program with a check of whether the execution of this program on a given platform satisfies a desired property. The platform is modelled through its digital clock. In [26], Halkjaer et al. use timed-automata modelling to identify that a particular scheduler (i.e. the legOS scheduler) suffers from starvation and shows that a revised design of the scheduler does not. In [31], Hendriks and Verhoef show that timed automata can be used to model and analyze timeliness properties of embedded systems architectures, by systematically modelling and analyzing a case study. However, this is done by constructing the timed-automata models manually. In [54], Ovatman et al. provide experiments using priced timed automata for schedulability analysis as well as analysis of resource consumption.

In [23, 22], Fersman et al. present decidability results for schedulability analysis using timed automata. The overall conclusion of this work is that the

schedulability-checking problem is undecidable if the following three conditions hold: 1) execution times are intervals; 2) the precise finishing time of a task instance influences other task releases; and 3) preemption is allowed. In [39], Krcál and Yi show that if one of the three conditions is dropped, the problem is decidable. Based on these results and the work on the UPPAAL model checker, Uppsala University has released the Times Tool [6, 21], which is a tool set for modelling, schedulability analysis and synthesis of schedulers and executable code. The version of Times Tool available at the current time is directly applicable for single-processor systems only.

In [19], David et al. provide alternative timed-automata implementations based on the models provided in [9]. The implementations generalize in certain areas, e.g. to include notion of jitter in task releases. The authors claim to provide an "...alternative account on how to model multiprocessor-scheduling scenarios most efficiently, by making full use of the modeling formalism of UPPAAL". It is unclear how this is most efficient and there are no examples verified that could substantiate this claim. The models provided in [9] are similar to the ones given in this dissertation in Section 5.1.6. The authors of [19] do not relate to stopwatch models like the ones in Sections 5.1.4 and 5.1.5 or to genuine discrete models like the one in Section 5.1.7. It turns out that most types of analysis are verified more efficiently on the basis of such models.

1.1.4 Simulation-based analysis

Simulation-based methodologies are still the predominant technique for performance evaluation of embedded systems. In simulation-based approaches SystemC [41] is a de-facto language for system modelling. SystemC is a C++ library that supports modelling, which is true to the underlying hardware and provides an executable model that can be used for simulation. These SystemC-based simulation approaches can target either a specific area (e.g. communication or the real-time operating system) or provide system-level analysis. In system-level analysis, the full embedded system is modelled from application to execution platform, also modelled is the mapping of the application onto the platform as well as the communication.

1.1.4.1 Specific-area analysis

In [44], Loghi et al. present MPARM, a cycle-accurate and signal-accurate analysis of on-chip communication in a MPSoC environment. The interconnects are

simulated and different architectures are compared. The processors in the system are modelled by instruction-set simulators and all hardware is coded in SystemC.

There have been several examples of using simulation-based SystemC models for analysis specifically of the real-time operating system (RTOS). In [33], Hessel et al. provide an abstract RTOS model for embedded systems. The model includes a task model, a scheduler and synchronization. The goal of this work is to minimize the number of context switches. In [52], Moigne et al. present a generic RTOS model. The model includes synchronization, message parsing and global data sharing. The work focuses on durations of scheduling, context loads and context saves.

1.1.4.2 System-level analysis

Within simulation-based system-level analysis much work has been conducted. Most of the modelling done at system level is based on the Y-chart of system design [57]. In the Y-chart there is a clear distinction between the application and the execution platform, and there is an explicit mapping of elements of the application onto the different parts of the platform.

In [38], Kempf et al. propose a SystemC-based simulation framework that allows evaluation of different mappings of the application onto the execution platform. The evaluation is conducted on an executable model of the system with annotated timing characteristics. The key element of this approach is a configurable event-driven virtual processing unit, which captures timing behaviors of the platform.

In [30], Haubelt et al. present a SystemC-based design methodology for mixed hardware/software solutions mapped to FPGA-based platforms. The approach supports automatic design space exploration, automatic performance evaluation and automatic system generation. The core result of this work is SysteMoC, a SystemC-based library that permits execution of well-known models of computation, which have been applied in the design of digital signal processing algorithms.

In [56], Pimentel et al. present the Sesame framework, which provides high-level modelling and simulation methods as well as tools for system-level performance

evaluation and exploration of heterogeneous embedded systems. Models of both the application and the platform are represented by graphs annotated with performance characteristics, e.g. computation requirements for each node in the application graph and processing capacity and power consumption for each node in the platform. In the exploration, the authors use the Strength Pareto Evolutionary Algorithm to find sets of approximated Pareto-optimal mapping solutions, i.e. solutions that are not dominated in terms of quality (performance, power and cost) by any other solution in the feasible set. SystemC simulation is used to provide performance estimates for the candidate solutions.

In [48, 47, 50], Madsen et al. present the SystemC-based framework ARTS. ARTS allows designers of multiprocessor system-on-chips to explore and analyze network performance, consequences of different mappings including memory and power usage and effects of RTOS selections including scheduling, synchronization and resource allocation policies. We will base our work on ARTS and will elaborate on this approach in Chapter 2.

1.1.5 Summary of different analysis approaches

Classic scheduling theory works well for single processor systems under rather ideal assumptions. However, for larger multiprocessor systems, the classic approach lacks structure and compositionality, which dramatically increases the complexity of the equations in the underlying analysis. Although some approaches for including properties of the platform (e.g. communication strategies as in [58]) have been examined, actual system-level analysis - where all levels of the system are analyzed - is not. Also, the classic scheduling theory is based on pretty idealized models, e.g. although issues regarding allocation of shared resources are addressed, there is no concept of dependencies among tasks.

The event-stream analysis approaches introduces structure and compositionality. It does this through an over-approximation. Although this approach yields results for systems of very large size, the over-approximation can also in some cases lead to very pessimistic analyses.

The timed-automata approaches generally has two down falls. Many timed-automata-based analyses are very case oriented. This means that a very specific system is analyzed, but there is little (if any) automation and no clear underlying formal model. If any other system is to be analyzed, it must be manually modelled from scratch. Other timed-automata approaches such as Times Tool

become very general. The analysis becomes mostly an analysis of the application as very little of the platform is modelled.

The simulation-based approaches give valuable input to designers early in the design process. However, as these approaches only examine some of the state space of the system, these approaches cannot provide guarantees. On the other hand, some of these simulation-based approaches, in particular ARTS, capture the generic structures and provide a modelling terminology that is very useful in embedded systems analysis. This terminology just lacks a clear formal model as a basis. In Chapter 4, we will formalize this terminology.

1.2 Motivation

The work described in this dissertation is intended to help designers of embedded systems. The aim is to provide designers with tools, models, languages, methodologies, etc., that in early stages of the design process can help the designer analyze different configurations and setups of systems.

Embedded systems today are getting more complex. It is difficult to use traditional methods to design, verify, validate and test them to make sure they are correct, reliable and not over-dimensioned. Over-dimensioning is when a designer would use larger, faster, more expensive, etc. components in systems in order to be "on the safe side". Results of over-dimensioning are usually waste of energy, space, money, etc.

Designs can also be oversimplified due to the complexity and lack of analysis methods. Oversimplification could be when a system is divided up into several individual parts that do not interact (e.g. an automotive embedded system where the airbag system is isolated from the rest of the embedded system). This oversimplification could result in limited functionality (e.g. lack of communication between systems) or in redundant subcomponents such as several identical sensors one for each individual part of the system in order to avoid interaction.

Since embedded systems are growing dramatically in complexity, they can do more and have more internal actions between very different components. This makes it hard to have an overview and to know what areas are critical and then test them. Many embedded systems are safety critical, and faulty systems could have fatal consequences.

When able to verify complex systems, one can:

1. Make bigger, life-critical systems that have more functionality, correctness and reliability. This can result in being able to solve bigger and more interesting problems while guaranteeing certain properties of the system.

2. Avoid over-dimensioning by creating less wasteful systems that are cheaper, smaller and more energy efficient.

3. Employ a systematic approach based on formal methods. This allows for a systematic design process in which one can easily re-verify system properties when making small alterations early in the design phases.

1.2.1 An embedded system - windmill control

Consider an embedded control application for a windmill - this has deliberately been chosen as an academic example (not as a realistic real-life control system), in order to provide a more intuitive understanding of concepts. The windmill has an anemometer that measures wind speeds and a windvane that measures wind direction. The purpose of the embedded control application is a) to point the windmill as close as possible to the current wind direction and b) for safety reasons, to deploy a brake to stop the motion of the windmill when wind speeds exceed some threshold (e.g. 30 m/s). In order to meet these purposes, four individual tasks (τ_1, τ_2, τ_3 and τ_4) of the application can be identified:

τ_1 Take a reading from the anemometer and add it to a list of the 10 latest wind speed readings - every 4 milliseconds

τ_2 Take a reading from the windvane and compare that direction to the direction that the windmill is currently pointing. The result of this comparison is a message, which can be represented in 2 bits - every 6 milliseconds

τ_3 Based on the current comparison (conducted by τ_2), turn the windmill toward the direction of the wind. In order for τ_3 to begin execution, τ_2 must have finished its execution. We say that there is a dependency from τ_2 to τ_3. - every 6 milliseconds

τ_4 While the windmill is running, if three or more of the 10 latest wind speed readings exceed the threshold, employ the brake to stop the windmill. While the windmill is not running (i.e. the brake is employed), if all of the 10 latest wind speed readings are below the threshold, release the brake to start the windmill. This task should not be executed before 10 readings are available (i.e. 40 milliseconds after the system starts). These checks are to be conducted every 6 milliseconds

These four tasks are to be executed on an execution platform. Consider an execution platform made up of two processing elements (pe_1 and pe_2). The operating systems on both of these, os_1 and os_2, can schedule tasks mapped to them based on a scheduling principle, either rate-monotonic or earliest-deadline-first. Initially rate-monotonic scheduling is chosen on both. In order to communicate messages from one to the other processing element resulting from inter-processor dependencies - i.e. when tasks in a dependency are mapped to different processing elements - the processing elements are connected via a bus (b_1). This bus can transfer 2 bits/millisecond and uses a first-in first-out (FIFO) arbiter.

The mapping of the application onto the execution platform is as follows: τ_1 and τ_2 are mapped to pe_1, whereas τ_3 and τ_4 are mapped to pe_2. Note that since τ_2 and τ_3 are mapped to different processing elements, their dependency require the data (2 bits) to be transferred on b_1 (at speed 2 bits/millisecond) - obviously this transfer takes 1 millisecond. In Figure 1.4, a graphical presentation of this system is given.

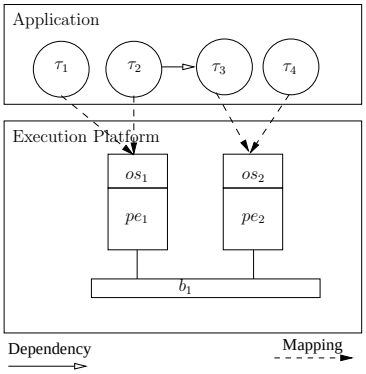


Figure 1.4: A windmill control system

The best-case execution time ($bcet$) and worst-case execution time ($wcet$) in milliseconds of executing the individual tasks of the application on the processing elements are given as $(bcet, wcet)$ here:

	τ_1	τ_2	τ_3	τ_4
pe_1	$(2, 2)$	$(1, 1)$	$(5, 5)$	
pe_2			$(2, 2)$	$(2, 3)$

Note that τ_3 can be executed by both processing elements, but execution on pe_2 is much faster. Note also that for all tasks other than τ_4, *bcet* and *wcet* are the same - i.e. execution of these tasks takes the same amount of time each time they are executed. However, τ_4 can be executed in 2 milliseconds in best case, that is, if no action is needed (i.e. the brake need not be deployed or released), whereas execution takes 3 milliseconds in worst case, if action is needed.

It is worth noting here, that in other approaches such as RTC [61] and SymTA [32] the notion of jitter is used to analyze and explain the issues that occur when dependent tasks mapped to different processing elements have a difference between best-case execution time and worst-case execution time. In analysis using jitter, the individual parts of the system are analyzed separately, and jitter propagates the uncertainty of tasks finishing times. In the analysis explored here, we will not use the notion of jitter. Instead, we consider all possible traces of the system in question, beginning from the initial start of the system.

The general question we would like an answer to, is whether or not the system will be able to autonomously execute the tasks forever (or at least infinitely long). In other words, will the chosen settings and properties make the overall system schedulable? If the answer is no, it would be desirable to have evidence of a situation where the system is not able to execute some task, i.e. where the system deadlocks.

Actual schedulability analysis of this system will reveal that the system can miss deadlines. If the platform is modified slightly, by choosing earliest-deadline-first as scheduling principle for the operating system os_2, the system will pass schedulability analysis without revealing deadline misses.

The windmill control system shown here is deliberately chosen as an academic example. We do not conclude that a windmill control system should be executed on a platform consisting of two processing elements as described here or that the exact timing properties are realistic. The example only highlights the terminology used and provides more intuitive understanding of some of the concepts involved when specifying embedded systems.

1.3 Purpose of this Project

In this project we aim at a concrete syntax and semantics that are based on ARTS models. These models are suitable for specification and analysis of systems early in the design process, at a stage where application and execution platform can be identified, but where neither are implemented or fully produced.

The application is characterized by only a task graph and timing requirements but without any detailed implementation details. The execution platform is defined simply in terms of number of processing elements, their general operating system properties and the interconnects, again without specific details on concrete implementation or synthesis.

Analysis at this stage gives the designer opportunity to make important design decisions without paying for decisions being made at later stages, when more implementation detail has been decided on, or when parts of the system have been developed or synthesized.

The thesis of our work is two fold:

1. That semantically-based verification of embedded systems can be suitable for problems that resemble industrially interesting examples of systems in size and complexity.

2. That models of timed automata, together with tools (specifically UPPAAL), are valuable as implementation platforms and verification backends for verifying properties of systems modelled.

1.4 The Structure of the Dissertation

The structure of the dissertation is as follows:

Chapter 2: We elaborate on the approach used in ARTS. The different components of ARTS specifications are explained and the underlying model used when simulating is discussed.

Chapter 3: We provide a language for specification of systems at an early point in the design process. This point is when the following three parts have been identified: 1) an application consisting of a number of interacting tasks , 2) an

execution platform made up of a number of interconnected processing elements and 3) a mapping of the tasks onto the processing elements.

Chapter 4: We derive a formal model that captures the different aspects of embedded systems and, in particular, formalizes the schedulability problem for embedded systems.

Chapter 5: We provide examples of how this formal model can be used as a basis for analysis and, in particular, how model checking of timed automata models implementing the formal model can be used to verify the schedulability problem.

Chapter 6: The tool MoVES is presented. MoVES can analyze systems specified in the language and, using an implementation of the formal model, make automatic verification of the schedulability problem possible.

Chapter 7: We give a range of different examples of systems, their specifications, their interesting properties and explanations of analysis using MoVES.

Chapter 8: We give indications to how this work can be further developed and what some of the consequences of early phase modelling have. We also show how the abstract models relate to models and implementations much closer to a final product.

Chapter 2

ARTS Concepts and Informal Model

This chapter will introduce the concepts used in ARTS, a multi-processor system-on-chip (MPSoC) simulation framework developed at the Technical University of Denmark. There will also be explanations of the underlying informal model. ARTS is a SystemC-based framework that is constructed to make it easy for developers to try out different setups early on in the design process. It provides a simulation engine that can assist in evaluating crosslayer causality between the application, the operating system and the platform architecture. ARTS is a system-level framework, which can be identified as a framework for the overall system comprising the application, real-time operating system and the execution platform. It has been a conscious design decision that the simulation engine is clearly separated from the model.

The ARTS framework models the *application* (described as *task* graphs capturing *dependencies* among tasks), the *execution platform*, i.e. the *processing elements*, their *interconnects* and the overall architecture and finally the *mapping* of the tasks onto the processing elements. In Figure 2.1 these components are depicted.

A SystemC implementation of the models is then instantiated and can simulate the system for a given amount of time. The result of the simulation is a runtime profile that allows the designer to evaluate the modelled system. The runtime

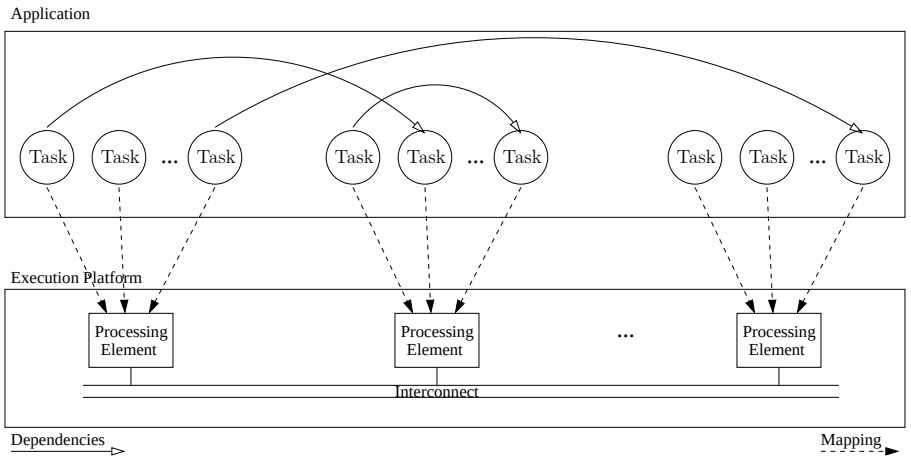


Figure 2.1: ARTS components

profile includes processor utilization, task response times and possible deadline misses, as well as memory-, communication-, and power-usage profiles. In the following sections the ingredients of ARTS are introduced.

2.1 Application

An application in ARTS is characterized by task graphs with a number of individual tasks and their data *dependencies*. Figure 2.2 shows a task graph for an example of a real-life application, an MP3 decoder. The tasks in the top row of the figure (τ_1, τ_3, τ_5, τ_9, τ_{11}, τ_{13} and τ_{15}) operate on the right channel of a stereo signal, whereas the tasks in the bottom row (τ_2, τ_4, τ_6, τ_8, τ_{10}, τ_{12} and τ_{14}) operate on the left channel. The tasks τ_0 and τ_7 are synchronization points from the MP3 decoder application. This example is examined and analyzed with ARTS in [48]. Each task is defined by the relative *deadline*, the *period*, an initial *offset* (or phase) and the *execution time*. Figure 2.3 gives an idea of what these terms mean on a timeline and each term is explained further here:

A Task dependency (the arrows in Figure 2.2) indicates that some task needs to finish its execution before another can start. That the task τ_2 is dependent on the task τ_1 (also written $\tau_1 \prec \tau_2$) means that τ_1 must finish executing before τ_2 can start. Each arrow in the task graph corresponds to such a dependency. In the case of dependencies between tasks mapped to different processing elements, the dependency may be defined with a size. This size indicates how large a mes-

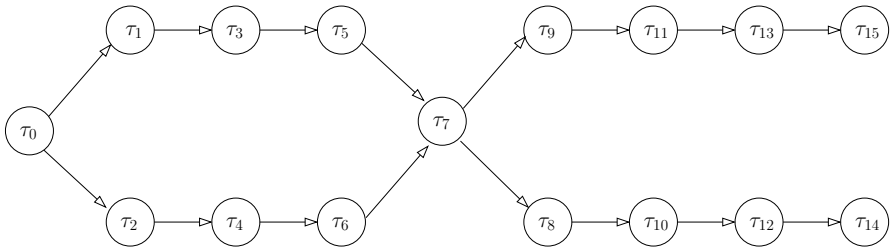


Figure 2.2: Task graph of an MP3 decoder

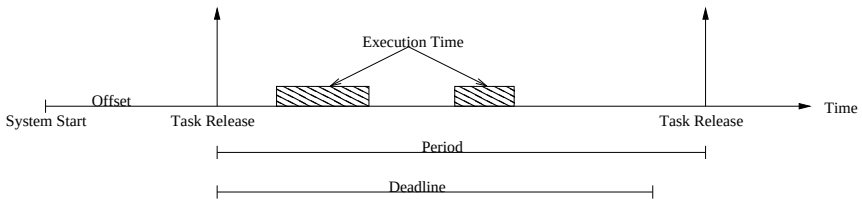


Figure 2.3: Timeline of a task from system start to its first period

sage is transferred as a result of the dependency. This transfer is done through the interconnects of the system; see more on interconnects in Section 2.2.2.

The relative deadline indicates how long after the release the task has to finish executing. If any deadline in any period of any task of the system is not met, the system is not schedulable.

The period of a task indicates how often the task is released, e.g. a period of 5 milliseconds for a task means that a new instance of the task is released every 5 milliseconds.

The offset (or phase) of a task defines when the first instance of the task is released relative to the start of the system, e.g. an offset of 3 milliseconds indicates that the first instance of the task is released 3 milliseconds after the start of the system.

The execution time of a task indicates how much processor time each instance of the task needs for execution, e.g. an execution time of 2 milliseconds means that the task needs 2 milliseconds of execution time each period. The execution time can be defined as an interval from best-case execution time (bcet) to worst-case execution time (wcet), meaning that each instance of the task needs an amount of execution time that is in this interval. Note that the execution time may depend on which processing element the task is mapped to.

The tasks considered within this terminology are cyclic tasks. This means that for each period a new instance of the task is released, and the task requires execution time in each period.

2.2 Execution Platform

The execution platform is characterized by the processing elements, their interconnects and the overall architecture of the system. Processing elements provide processing power to execute the tasks of the system. These processing elements can be either dedicated components that only execute one specific task (e.g. an application-specific integrated circuit (ASIC)), more general components that can execute many different tasks (e.g. a general purpose processor (GPP)), or even components that can do a few specific tasks (e.g. a field-programmable gate array (FPGA)). Figure 2.4 shows an example of an execution platform con-

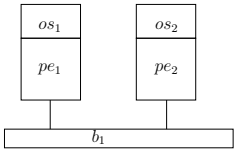


Figure 2.4: A platform example

sisting of the two processing elements pe_1 and pe_2, with the real-time operating systems os_1 and os_2, respectively. The two processing elements are connected via the shared bus b_1.

2.2.1 Processing elements

A processing element in ARTS is modelled by its real-time operating systems and has the following functionalities: a scheduler, a synchronizer and an allocator.

Scheduler

The scheduler will grant processing time to the tasks on the processing element. The scheduler can do this based on a priority-based scheduling principle, which means that whenever two or more tasks are ready to be executed on that pro-

cessing element, the task with highest priority is selected for execution.

Scheduling can be either preemptive or non-preemptive.

Preemptive scheduling is when the scheduler can temporarily stop (preempt) currently executing tasks to allow more important (higher prioritized) tasks to execute instead. This implies that scheduling can occur whenever a task becomes ready or when a task finishes and others are waiting for execution.

For *non-preemptive scheduling*, when a task has started execution, it will continue its execution until it finishes, regardless of whether tasks with higher priority are released. The implication of non-preemptive scheduling is that scheduling can only occur when a task finishes and others are ready for execution or when a task becomes ready and no other tasks are waiting.

A scheduling principle is either static or dynamic.

Static scheduling principle: A processing element running a static scheduling principle bases each scheduling decision on a pre-determined prioritized list of all tasks; this list can be made at compile-time. An example of a static scheduling principle is rate-monotonic (RM) scheduling, for which tasks with shorter periods have higher priority.

Dynamic scheduling principle: A processing element using a dynamic scheduling principle bases each scheduling decision on properties of the current situation. An example of a dynamic scheduling principle is earliest-deadline-first (EDF) scheduling, where tasks closer to their deadline have higher priority.

Synchronizer

The synchronizer manages task dependencies. When a task τ is released, the synchronizer determines whether the task τ depends on has already finished. When a task finishes, the synchronizer notifies any tasks that are dependent on it. Note that if dependant tasks are located on different processing elements, this notification must be communicated.

Allocator

The allocator manages access to shared resources; this could be shared busses, memories, i/o devices, etc. The allocator can implement a principle such as priority ceiling in order to avoid blocking. Blocking occurs when a lower prioritized task blocks access to a processing element through a shared resource. In the priority ceiling protocol, a lower prioritized task will inherit the priority of higher prioritized tasks for which it holds access to a resource shared between the tasks.

Note that certain properties of tasks may depend on which of the processing elements it is mapped to. For example, a task may run faster or slower on different processing elements, and some tasks may not be able to execute on certain processing elements at all. Also, when including resource usage such as power- and memory-usage in the analysis, the individual power- and memory-footprints for each task may differ when executed on different processing elements.

2.2.2 Interconnects

Interconnects are links between different processing elements. These interconnects are defined in terms of their speed, i.e. how fast they can deliver messages. Dependencies of tasks mapped to different processing elements (inter-processor dependencies) may need to have a message transferred, and these messages are given in terms of their size. The time that it takes to deliver the message is then determined by the speed of the link and the size of the message, e.g. a message with the size 4 bits delivered on a link with the speed 2 bits/millisecond will take 2 milliseconds to deliver. An example of such a link is the bus b_1 in Figure 2.4.

2.3 Mapping

Mapping of an application onto an execution platform is valuable to include in analysis. Through explicit mapping of different parts of the application onto the specific processing elements, inter-processor dependencies are identified and task-specific properties, ones that depend on which processing element they are mapped to, can be revealed. Inter-processor dependencies have the capability to introduce *multiprocessor anomalies*. A multiprocessor anomaly is a timing anomaly that occur when a local worst-case behavior does not give global worst-

case behavior. Including the mapping in analysis allows for identification of such multiprocessor anomalies.

The mapping defines which tasks are to be executed on which processing elements. This mapping may have greater implications than one realizes. Some tasks may be able to be executed on different processing elements, but they may have different characteristics, e.g. required execution time on different processing elements. Also, mapping dependent tasks to different processing elements may require transfer of messages over links. Finally, some processing elements may run different real-time operating systems, and the system performance may vary extensively when a task is mapped to one processing element as opposed to another.

Figure 2.5 shows the MP3 decoder from Figure 2.2 mapped onto the platform on Figure 2.4, where the tasks above the dotted line are mapped to pe_1 and the task below are mapped to pe_2. Note that the dependencies crossing the dotted line, i.e. $\tau_0 \prec \tau_1$, $\tau_6 \prec \tau_7$ and $\tau_7 \prec \tau_8$ are inter-processor dependencies and may require data transfer over the bus b_1 (see Figure 2.4). This is only one possible

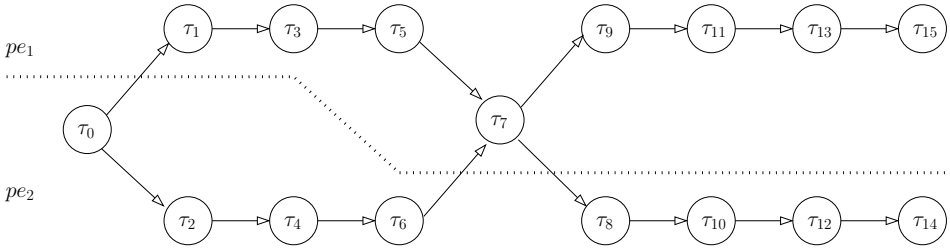


Figure 2.5: Mapping of an MP3 decoder onto a platform

mapping, and the system may behave quite differently if even a slightly different mapping is used.

2.4 Schedulability

Schedulability of a system defined through the ARTS terminology as described here is to be understood as follows: A system is schedulable if no deadline for any task in any period is ever missed. For systems where tasks have execution times as intervals from bcet to wcet, this means that for every period, all execution times in that interval should not lead to missed deadlines anywhere in the system. Note that it is not enough to just examine wcet as systems may

contain multi-processor anomalies. In such cases, execution times that are not wcet can trigger missed deadlines in the system.

2.5 Simulation

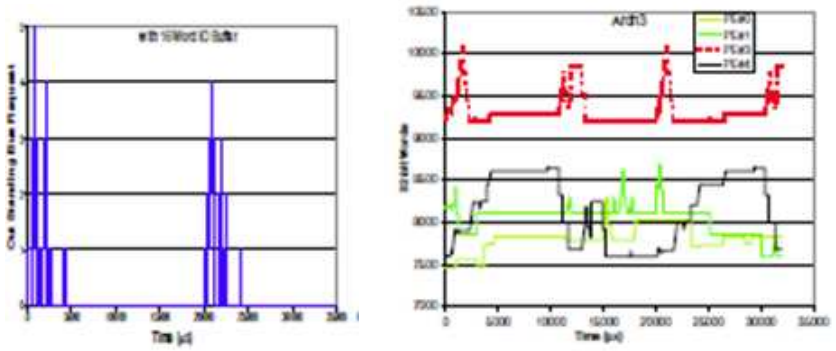
Once a system has been modelled through the aforementioned components, the system can be simulated. A simulation is conducted by letting the modelled components of the system run according to the properties specified. In the case of execution time as an interval, for each time the task is released, a random number between bcet and wcet is chosen. The duration of simulation is specified beforehand as a number of cycles. After this number of cycles the simulation is stopped and the results are available. These results can be valuable to a developer as he can try out setups of a system early in the design process. It gives an indication of how the system will act in the average case. But the simulation can give no guarantees about best- or worst-case performance.

The result of simulating a system modelled in ARTS is a runtime profile. This profile can include processor utilization, task response times and possible deadline misses, as well as memory-, communication-, and power-usage profiles of the modelled system for the simulated duration. In Figure 2.6 examples of simulation output from ARTS are given with profiles for bus contention (2.6(a)) and memory (2.6(b)). These profiles are part of examples published in [49]. Since this is a simulation examining only certain (random) traces of the system's execution, there is no guarantee that best- or worst-case performance of the system has been targeted. Therefore, this is a good guide for average case system performance, but not for giving guarantees on system behavior.

2.6 Summary

In this chapter we have provided an overview of the underlying model of the multi-processor system-on-chip simulation framework ARTS. This framework allows developers to try out different setups of the systems they wish to design early in the design process. Although informal, the underlying model has a clear structure with which the different components of a system are modelled individually and systematically.

An ARTS model of a system consists of an application mapped onto an execution platform. An application is made up of task graphs, including the tasks'



(a) Bus contention profile (b) Memory profile

Figure 2.6: Example of ARTS simulation results

dependencies, and an execution platform is modelled through the processing elements as well as their interconnects.

ARTS also provides a terminology that is useful when we look deeper into the model and give it semantics. A great advantage of the approach used in developing ARTS is that there has been a conscious decision to keep the simulation engine apart from the model. This makes the model much more suitable for establishing a formal semantics. It also keeps the model generic so there is a clear structure, which makes it easy to define new systems within the same framework.

Chapter 3

The MoVES Language

The exploration in Chapter 1 identified that most of the approaches in the area of analysis of embedded systems seem to fall into categories, where either a) analysis is done for only one specific part of the system so global effects on local choices are not analyzed, or b) analysis is based on informal or sparse models that make it hard to comprehend the full system and that only allow for sporadic analysis (simulation), thereby cannot give guarantees.

This chapter aims at providing a formal model for embedded systems. The model should contain clear and precise characteristics of all components (such as tasks, processing elements, operating systems, etc.) of a system as well as the system's architecture (how different components communicate and how they are connected physically). This structure is chosen in order to be able to conduct *cross-layer analysis*, i.e. analysis of systems across several layers (e.g. application layer, execution-platform layer and mapping layer). Only through this type of analysis, problems originating at one level that results in issues at another level, can be thoroughly analyzed. As a basis, this model uses the informal model and terminology defined by ARTS, which was explained in Chapter 2. One could say that it is an attempt to give semantics to ARTS models.

The model, which is defined in Chapter 4, captures the characteristics of each

individual component in the system and their communication. Furthermore, it keeps a clear structure that reflects the architecture of the system. Before reaching a formal model in Chapter 3, we will establish a concrete syntax for a language, *the MoVES language*, that can be used to specify systems that can be captured by such a model.

This chapter is based on the work initiated in [12], where a grammar was given. Here we extend the work and provide examples to help the understanding of the use and structure of the language.

3.1 Concrete Syntax for the MoVES Language

In order to specify an embedded system and the problem to be analyzed, a syntax should be established. Although many of the approaches mentioned in Section 1.1 provide languages or at least a specification structure with which systems can be expressed, most of these fail to capture each component individually and keep a clear structure that reflects the system architecture.

This section serves as the establishment of a syntax for a language, the MoVES language, which allows designers to express systems, consisting of applications executing on execution platforms and their mapping, through their individual components. At the same time the structure reflects the physical architecture of the system. The terminology used to capture individual components is identified from ARTS in Chapter 2, and the structure follows that of ARTS models as well.

3.1.1 Grammar

In Figure 3.1(a), the grammar for the MoVES language is given. This language is intended to be used to specify embedded systems and their analysis problems. In general, an embedded system is specified through a specification of its application (*app*), the execution platform (*plat*), a mapping (*map*) of the application onto the execution platform, computational requirements (*cr*) showing effects of different mappings and finally, the verification/validation property (*prop*) that should be examined.

```
system ::= app plat map cr prop
app ::= Application task+ dep
task ::= taskid per off
taskid ::= Task: tid
per ::= Period: n
off ::= Offset: n
dep ::= Dependencies dp*
dp ::= tid -> tid : n
plat ::= Platform proc+ bus
proc ::= Proc: pid Sch: sch
sch ::= FP | RM | EDF
bus ::= busid arbit speed
busid ::= Bus: bid
arbit ::= Arb: arb
arb ::= FIFO
speed ::= Speed: n
map ::= Mapping mp+
mp ::= tid : pid
cr ::= Creq tonp+
tonp ::= tid @ pid bcet wcet
bcet ::= Bcet: n
wcet ::= Wcet: n
prop ::= Property p
p ::= Schedule?
```

n ∈ ℕ, tid, pid and bid are strings,
terminal symbols are in **Roman**,
non-terminals are in *italics*

(a) MoVES Grammar

```
Application
 Task: T1
  Period: 4
  Offset: 0

 Task: T2
  Period: 6
  Offset: 0

 Task: T3
  Period: 6
  Offset: 0

 Task: T4
  Period: 6
  Offset: 40

 Dependencies
  T2 -> T3 : 2

Platform
 Proc: P1
  Sch: RM

 Proc: P2
  Sch: RM

 Bus: B1
  Arb: FIFO
  Speed: 2
```

```
Mapping
 T1 : P1
 T2 : P1
 T3 : P2
 T4 : P2

Creq
 T1 @ P1
  Bcet: 2
  Wcet: 2

 T2 @ P1
  Bcet: 1
  Wcet: 1

 T3 @ P1
  Bcet: 5
  Wcet: 5

 T3 @ P2
  Bcet: 2
  Wcet: 2

 T4 @ P2
  Bcet: 2
  Wcet: 3

Property
 Schedule?
```

(b) Windmill control specification

Figure 3.1: MoVES grammar and example specification

3.1.2 Example

In Figure 3.1(b), the specification of the windmill control system introduced in Section 1.2.1 is given using the MoVES language. This specification is given in two columns. The first column contains the application and platform parts. The second column contains the mapping, the computational requirements and the verification property. In the following sections this example will be used to describe the individual aspects of the MoVES language.

3.1.3 Application

In the example we see an application made up of four tasks `T1`, `T2`, `T3` and `T4`. The task `T1` has a period of 4 and the other tasks' period is 6. Task `T4` has an initial offset of 40, the other tasks do not have an offset. Finally, `T3` is dependant on `T2` and a message of size 2 needs to be transferred if these two tasks are mapped to different processing elements.

3.1.4 Execution platform

The platform consists of two processing elements `P1` and `P2`, both using rate-monotonic scheduling. They are connected to a bus `B1`, which has a first-in first-out arbiter and runs at speed 2 (e.g. a message of size 2 takes one time unit to transfer on this bus).

3.1.5 Mapping

The tasks `T1` and `T2` are mapped to `P1`. `T3` and `T4` are mapped to `P2`. Notice that there is an inter-processor dependency (T2 is mapped to P1 and T3 is mapped to P2).

3.1.6 Computational requirements

In the *computational requirements* for the example we see that the tasks `T1`, `T2` and `T3` can all be executed on `P1`, where `T1` requires 2 time units of execution time each period, `T2` requires 1 time unit and `T3` requires 5 time units, if mapped to `P1`. `T3` and `T4` can be executed on `P2`, where `T3` requires 2 time units of execution time each period and `T4` require 2 time units in best case and 3 time units in worst case each period.

Note that there should be an entry for each possible pair of tasks and processing elements, for which the task is executable on the processing element, even if they are not included in the actual mapping. This allows the designer to easily remap tasks without having to reformulate the computational requirements. For example, remapping `T3` to `P1` can be done without changing the application or the platform.

3.1.7 Verification/validation property

Finally, the verification/validation property is specified as `Schedule?` i.e. schedulability analysis. This analysis examines the schedulability of the specified system. In the event of non-schedulability, a trace resulting in a missed deadline is produced, i.e. a counter example to the schedulability problem.

3.1.8 On design space exploration

A conscious decision throughout the development of the syntax was that each component of the system should be specified individually, and that characteristics of the system that are dependent on the setup (e.g. the mapping of application onto execution platform) are specified separately of the application, execution platform and mapping. This decision allows for much easier and clearer *design space exploration*. Design space exploration should be understood as the process of revisiting the setup of the system in the design phase of development. In Figure 3.2, the basic ideas of design space exploration is depicted. This follows the structure of Y-chart-based design as used in [56]. Design space

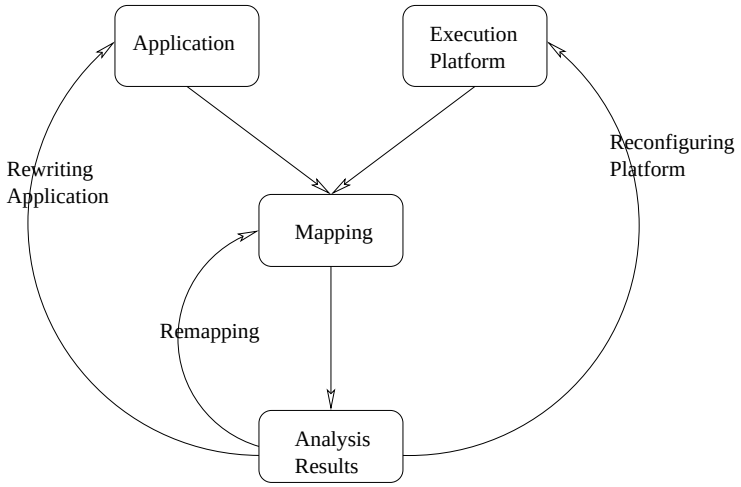


Figure 3.2: Design space exploration

exploration in this context constitutes the tasks of remapping, rewriting the application and reconfiguring the execution platform based on analysis results. This can be done in an iterative manner. Remapping occurs when some tasks are mapped to different processors. Rewriting the application can be e.g. redefining

the task graph or optimizing tasks to alter the execution times. Reconfiguring the execution platform can, for example, be connecting the processors differently or changing the operating system of the individual processing elements, e.g. changing scheduling principle. In the following, examples will highlight some of the features of design space exploration.

3.1.9 Testing schedulability

The result of the schedulability analysis of the system specified in Figure 3.1(b) will result in a missed deadline. A trace depicting the events leading up to this miss is shown in the diagram on Figure 3.3. In this trace it can be observed

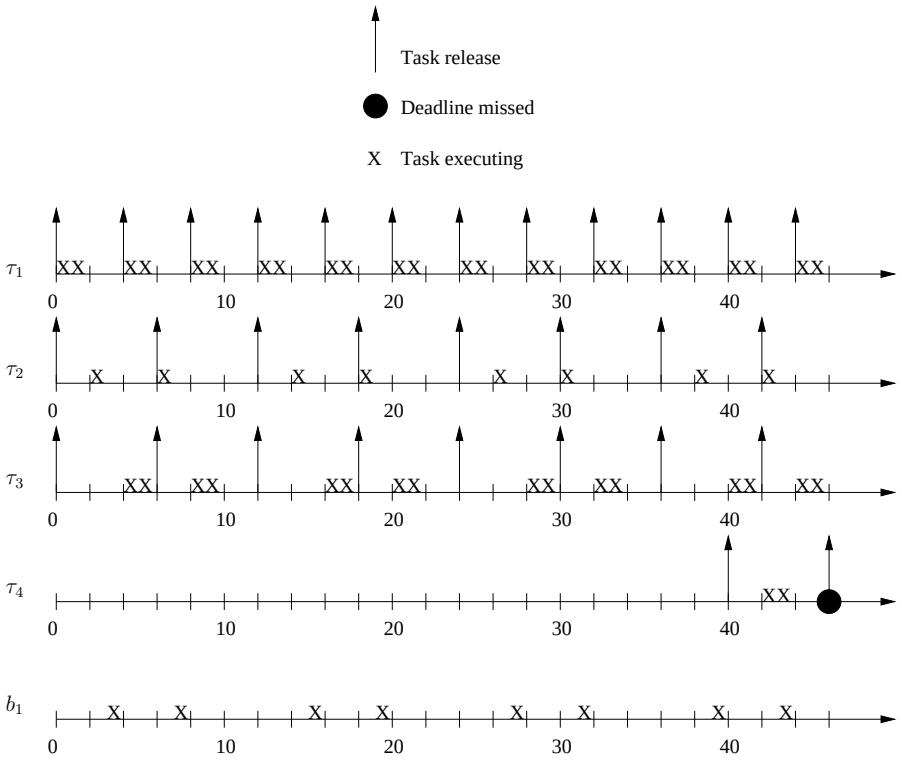


Figure 3.3: Missed deadline trace in schedulability analysis of windmill control system

that τ_4 misses a deadline after 46 milliseconds.

3.1.10 Exploring design decisions

In order to avoid the undesired behavior of non-schedulability of the windmill control system, the designer can explore different design decisions. In this case, if the designer wishes to change the scheduling principle on pe_2 to use earliest-deadline-first instead of rate-monotonic, this change can be done without making any other changes to the specification than replacing `RM` with `EDF` for pe_2. Analysis of the changed system reveals that the system is schedulable when setup like this.

Although this is just a small academic example of design space exploration, it gives an idea of which types of exploration can be done on the basis of the syntax derived at here. Even in this simple example, it is tedious for a human to spot the missed deadline.

It is also worth noting that if all offsets in this original specification were reduced to zero, the system would be schedulable. This is an example of a system where the notion of *critical instant* from classic scheduling theory does not invoke all missed deadlines, and therefore, schedulability analysis based on the critical instant does not hold. In [42], the notion of a critical instant is defined as occurring whenever a task is requested simultaneously with requests for all higher priority tasks. It is then proven that the worst-case response time occurs when task phasing creates a critical instant, i.e. when all offsets are zero. However, this example shows that multiprocessor systems can have worst-case response time even when not when all offsets are zero.

3.1.11 Exploring multiprocessor anomalies

In Figure 3.4 a small academic example system is graphically depicted and in Figure 3.5(a) a specification of this system is provided.

When conducting schedulability analysis of this system, it is encountered that the system is not schedulable. The trace in Figure 3.5(b) shows how a missed deadline is triggered.

When altering the specification to let the task `T1` execute only in worst case (i.e. 2) and conducting analysis on that system, an interesting result is found. It turns out that the system specified as such is schedulable. In other words, only including worst-case execution times locally for each task does not reveal the worst-case globally, i.e. the specified system possesses a multiprocessor

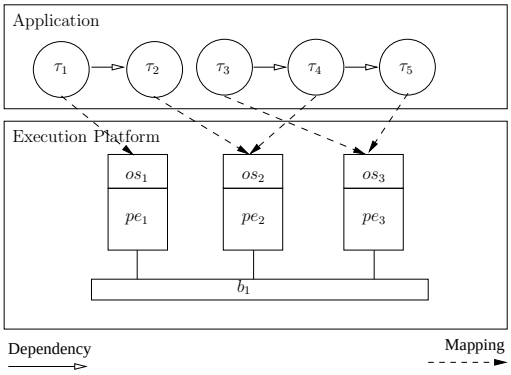


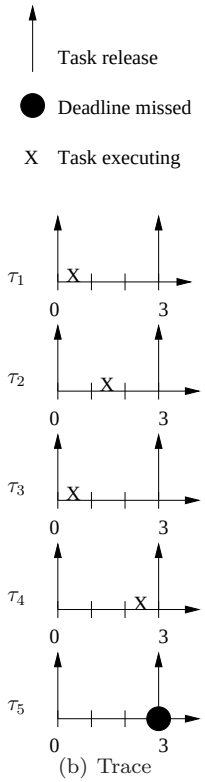
Figure 3.4: Small academic example system

```
Application
 Task: T1
  Period: 3
  Offset: 0

 Task: T2
  Period: 3
  Offset: 0

 Task: T3
  Period: 3
  Offset: 0

 Task: T4
  Period: 3
  Offset: 0

 Task: T5
  Period: 3
  Offset: 0

 Dependencies
  T1 -> T2 : 0
  T3 -> T4 : 0
  T4 -> T5 : 0
```

(a) Specification

```
Platform
 Proc: P1
  Sch: RM

 Proc: P2
  Sch: RM

 Proc: P3
  Sch: RM

 Bus: B1
  Arb: FIFO
  Speed: 1

Mapping
 T1 : P1
 T2 : P2
 T3 : P3
 T4 : P2
 T5 : P3
```

```
Creq
 T1 @ P1
  Bcet: 1
  Wcet: 2

 T2 @ P2
  Bcet: 1
  Wcet: 1

 T3 @ P3
  Bcet: 1
  Wcet: 1

 T4 @ P2
  Bcet: 1
  Wcet: 1

 T5 @ P3
  Bcet: 1
  Wcet: 1

Property
 Schedule?
```



(b) Trace

Figure 3.5: MoVES specification and missed deadline trace for system with multiprocessor anomaly

anomaly. Even in this small academic example, the anomaly is not easily found. This provides even more evidence that analysis techniques which can reveal these are needed.

3.2 Summary

In this section we have defined the MoVES language. This language captures the relevant aspects of embedded systems and follows the terminology and structure of ARTS as described in Chapter 2. An example showed that in the language there is a separation of concerns, i.e. aspects of the application, the platform and the mapping are all individually specified. This allows for structured design space exploration since a small chance in one area of the system only require altering in the specification of that specific area.

Also, the examples show that single-processor scheduling theory does not generalize to the multiprocessor domain when data dependencies are taken into account due to the notion of multiprocessor anomalies. The use of the notion of the *critical instant* does not always give a valid analysis, as some multiprocessor systems (e.g. the windmill control system) are schedulable without offsets but not with. Also, for some systems, what is best case locally can trigger worst case globally, as was seen in the example in Section 3.1.11.

Chapter 4

Semantics for MoVES

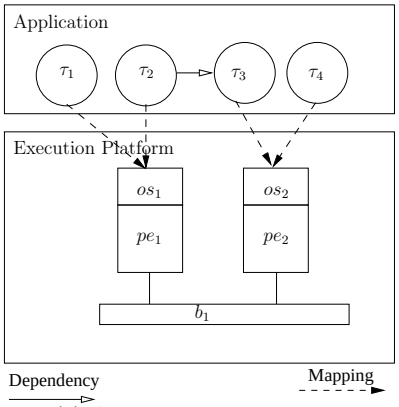
In this chapter, we will give a model for systems such as the ones described in the previous chapters. Firstly, the key concepts of the semantics will be explained informally, after which each of the individual parts of the model will be given formally: the application, the platform, the mapping and scheduling of tasks. Then the model of computation is formalized, and finally, decidability of schedulability analysis problem is determined.

The chapter is based on the work presented in [9] with an informal explanation of the concepts involved to promote readability.

4.1 Semantical Concepts Explained Informally

As the formalization in this chapter at times can get very detailed, this section will serve as an informal explanation of some of the concepts and the basic idea of analysis. We will rely on the example of the windmill control system, which was explained in Section 1.2.1, in order to provide this informal explanation. Figure 4.1 serves as a reminder of this example.

A system consists of an application running on an execution platform. The application can be divided up into a number of tasks. The windmill control



(a) Application and Platform

	τ_1	τ_2	τ_3	τ_4
pe_1	$(2,2)$	$(1,1)$	$(5,5)$	
pe_2			$(2,2)$	$(2,3)$

(b) Computational requirements

Figure 4.1: Windmill control system

system consists of four tasks (τ_1, τ_2, τ_3 and τ_4), and these might have timing constraints (τ_1 has a period of 4 whereas the other tasks have a period of 6, and only τ_4 has an offset of 40, the other tasks have no offsets) as well as dependencies with other tasks, exemplified here by $\tau_2 \prec \tau_3$. The execution platform is made up of a number of programmable or dedicated processing elements, here pe_1 and pe_2. Each task is mapped onto a processing element. We write $\mathcal{T}_{pe_1}$ denoting the tasks mapped to pe_1, i.e. $\{\tau_1, \tau_2\}$ and $\mathcal{T}_{pe_2} = \{\tau_3, \tau_4\}$

Once the mapping is established, the computational requirements (i.e. best-case execution time ($bcet$) and worst-case execution time ($wcet$)) for each task of the system can be identified; in this case we have the following:

$$\begin{aligned} bcet_{\tau_1} &= wcet_{\tau_1} = 2 \\ bcet_{\tau_2} &= wcet_{\tau_2} = 1 \\ bcet_{\tau_3} &= wcet_{\tau_3} = 2 \\ bcet_{\tau_4} &= 2 \\ wcet_{\tau_4} &= 3 \end{aligned}$$

Processing elements that can execute several different tasks and grant these execution time dynamically require a dedicated real-time operating system (os) as a layer between the application and the execution platform. The real-time operating system manages scheduling of tasks as well as any dependencies the tasks have with each other. We formalize the concepts of three specific scheduling principles: fixed-priority, rate-monotonic and earliest-deadline-first, as relations among tasks.

The relations for fixed-priority ($>_{\mathrm{FP}}$) and rate-monotonic ($>_{\mathrm{RM}}$) are straightforward, as they are static and do not evolve over time. In the windmill control system, rate-monotonic scheduling is chosen on both pe_1 and pe_2. This means that τ_1 has the highest priority of all tasks since it has the shortest period. For the relation to be a total order, we use the numbering used to name the tasks, e.g. τ_2 has number 2, τ_3 has number 3, etc. This gives the following total order for the ($>_{\mathrm{RM}}$) relation: $\tau_1 >_{\mathrm{RM}} \tau_2 >_{\mathrm{RM}} \tau_3 >_{\mathrm{RM}} \tau_4$.

The relation for earliest-deadline-first ($>_{\mathrm{EDF}}$), however, requires more detail since the relation evolves over time to indicate which tasks are closest to their next deadline (here, the start of their next period). $\tau >^{t}_{\mathrm{EDF}} \tau'$ denotes that τ has higher earliest-deadline-first priority than τ' at time point t. To capture this "closeness" to the next deadline, we introduce $\mathrm{dist}_{\tau}(t)$ denoting the distance from time point t to the nearest deadline of τ.

Basically, the semantics of the computational model for MoVES revolves around the notion of a *state*. A state is a snapshot of what is happening on each processing element. For the processing element pr_i, the state element (s_i, ϵ_i) records which task it is currently executing (s_i) (if any, $\perp$ denotes that no tasks are currently executing) as well the processing element's current *execution vector* (ϵ_i). The execution vector ϵ_i captures how much execution time is needed by each task mapped to pe_i to finish their execution in the current period. If the task τ is mapped to pe_i, then $\epsilon_i(\tau)$ captures how much execution time is needed by τ to finish its execution in the current period. Whenever a new period for τ starts, a possibly non-deterministic choice of $\{bcet_{\tau}, \ldots, wcet_{\tau}\}$ is taken to choose the execution time for τ in that period. If *bcet* and *wcet* for a task is the same, then the choice of course is deterministic.

A *system state* is an M-tuple $\sigma = ((s_1, \epsilon_1), \ldots, (s_M, \epsilon_M))$ with an element for each of the M processing elements of the system. The initial system state for the windmill control system σ_1 is the following:

$$\sigma_1 = ((\tau_1, \langle 2, 1\rangle), (\perp, \langle 2, 0\rangle))$$

We see that pe_1 is executing τ_1 and the two tasks τ_1 and τ_2 mapped to pe_1 require 2 and 1 time units of execution, respectively, to finish in the current period. On pe_2 no task is executing and the two tasks τ_3 and τ_4 mapped to pe_2 require 2 and 0 (zero) time units, respectively to finish in the current period. The zero time units needed by τ_4 indicates that the task has not yet been released for its first period, i.e. it is in its offset phase.

If we look at the system after 40 time units, i.e. when τ_4 is released for the first time, the current system state is one of the following σ_{41a} or σ_{41b}:

$$\sigma_{41a} = ((\tau_1, \langle 2, 1\rangle), (\tau_3, \langle 2, 2\rangle)) \qquad \sigma_{41b} = ((\tau_1, \langle 2, 1\rangle), (\tau_3, \langle 2, 3\rangle))$$

Note that since, for τ_4, $(bcet = 2) \neq (wcet = 3)$, a non-deterministic choice between the choices for execution time is taken. This is the case each time a task with $bcet \neq wcet$ is released.

In Figure 4.2, the intuition of the finitely branching, infinite computation tree behind the model of the windmill control system is shown.

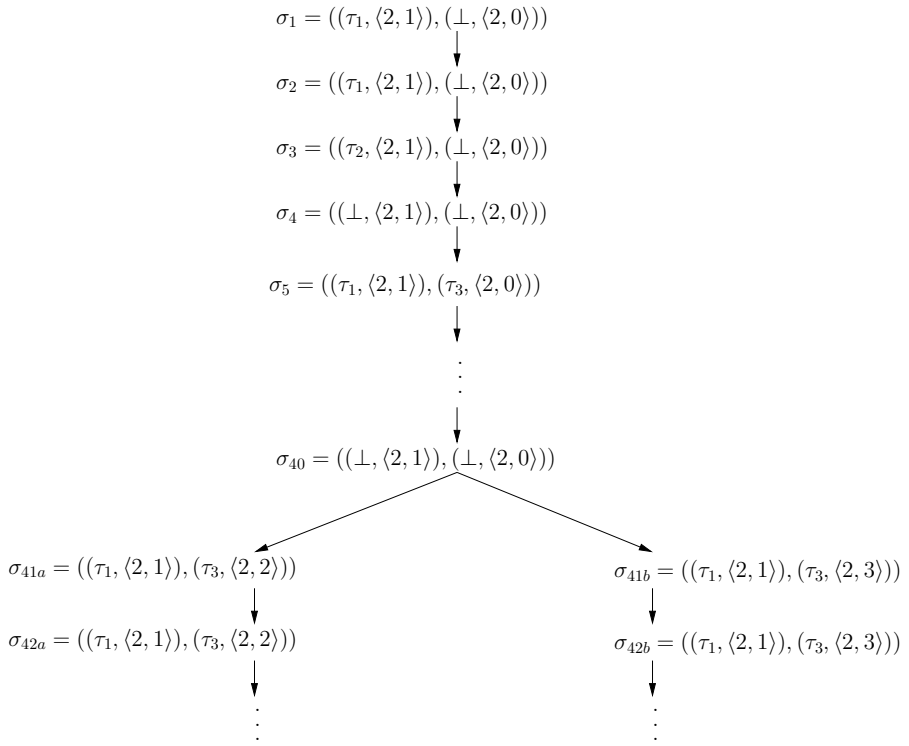


Figure 4.2: Computation tree for windmill control system

Although this tree is infinite, from each node in the tree, there is a finite sequence of states leading back to the initial state σ_1. We call such a sequence a *trace* ($\hbar$), e.g. the trace leading up to σ_{40} consists of the states $\sigma_1\sigma_2 \ldots \sigma_{40}$. The tree can be created by using the $Next_\hbar$, which defines the set of possible next system states by selecting the task to be executed (i.e. the task with highest priority given a scheduling principle), and determining the possible execution vectors for the next states, e.g. $Next_\hbar$ of the trace leading up to σ_{40} are $\{\sigma_{41a}, \sigma_{41b}\}$.

This has been an informal introduction to some of the concepts in the formalization. In the following sections, these concepts are formalized and semantics for applications executing on platforms is provided.

4.2 Application Model

Let a finite set $\mathcal{T}$ of *tasks* be given. Each task $\tau \in \mathcal{T}$ is characterized by a *period* $\pi_\tau \in \mathbb{N}$, a *best-case execution time* $bcet_\tau \in \mathbb{N}$, and a *worst-case execution time* $wcet_\tau \in \mathbb{N}$, where $wcet_\tau \geq bcet_\tau > 0$. A task τ needs a certain amount, between $bcet_\tau$ and $wcet_\tau$, of time units of a processor's time to finish its job in a given period. We formalize the notion of a processor's time later in Section 4.3. A task τ is characterized by an *initial offset* $o_\tau \in \mathbb{N}$, which means that the first period of τ starts o_τ time units after the system has started. Thus, the n'th period of task τ is the time interval $[o_\tau + (n-1) \cdot \pi_\tau, o_\tau + n \cdot \pi_\tau[\subset \mathbb{R}_{\geq 0}$, for $n = 1, 2, \ldots$. If the initial offset of a task is 0 (zero), then the task starts at system start.

An application is modelled by a *task graph* $G = (\mathcal{T}, \prec)$, where $\prec \subseteq \mathcal{T} \times \mathcal{T}$ is a directed, acyclic graph. An edge $(\tau, \tau') \in \prec$ (also written $\tau \prec \tau'$) represents a causal dependency, i.e. τ must finish its job before τ' can start.

A *sequential component* $G_s = (\mathcal{T}_s, \prec_s)$, where $\mathcal{T}_s \subseteq \mathcal{T}$ and $\prec_s \subseteq \mathcal{T}_s \times \mathcal{T}_s$, is a connected sub-graph of G, for which

- all tasks have the same period $\pi_{\mathcal{T}_s}$, i.e. $\pi_i = \pi_j = \pi_{\mathcal{T}_s}$, for $\tau_i, \tau_j \in \mathcal{T}_s$, and
- the offsets of tasks are so close to each other that their first (and hence the n'th) period overlaps, i.e. $|o_i - o_j| < \pi_{\mathcal{T}_s}$, for $\tau_i, \tau_j \in \mathcal{T}_s$.

We shall, from now on, assume that G can be partitioned into sequential components $G_i = (\mathcal{T}_i, \prec_i)$, for $i = 1, \ldots, N$, where $\mathcal{T}_i \cap \mathcal{T}_j = \emptyset$, whenever $i \neq j$, $\mathcal{T} = \mathcal{T}_1 \cup \cdots \cup \mathcal{T}_N$ and $\prec = \prec_1 \cup \cdots \cup \prec_N$.

Each sequential component of a task graph constitutes a model of an independent stream-processing program. In the case of a smart phone with capabilities for both GSM encoding and decoding as well as mp3 decoding, the task graph could for example be two sequential components, one for the GSM en-/decoding and another for the mp3 decoding. Note that these two components can have different periods.

4.3 Model of the Execution Platform

A *platform* consists of $M \geq 1$ *processing elements* (also called *processors*) $PE = (pe_1, pe_2, \ldots, pe_M)$, where each processing element pe_i is capable of executing

tasks according to a given *scheduling principle*. We will here consider the following preemptive scheduling strategies: rate-monotonic RM, fixed-priority FP and earliest-deadline-first EDF scheduling. There will be no principal difficulties in extending the model with more scheduling principles; but this will, of course, add more details.

4.4 Mapping (System Model)

A *system* consists of a *mapping* $m : \mathcal{T} \to \{1, \ldots, M\}$ of tasks to processing elements, and a *configuration* $Sch : PE \to \{\mathrm{RM}, \mathrm{FP}, \mathrm{EDF}\}$ of the scheduling principle used by each processing element. We shall often consider the set of tasks $\mathcal{T}_{pe_i}$ mapped to a particular processing element pe_i, i.e. let

$$\mathcal{T}_{pe_i} = m^{-1}(i) = \{\tau \in \mathcal{T} \mid m(\tau) = i\}$$

4.5 Scheduling of Tasks

We assume that each task has a unique number given by an injective function $pr : \mathcal{T} \to \{1, \ldots, \mathrm{card}(\mathcal{T})\}$. This numbering is used to define total orderings of tasks in connection with the various scheduling principles.

A task τ has *higher fixed priority than* τ', written $\tau >_{\mathrm{FP}} \tau'$, iff. $pr(\tau) < pr(\tau')$, i.e. smaller number given by pr means higher priority.

For rate-monotonic scheduling, the priority relation among tasks is primarily given by their periods (shorter periods mean higher priority) and secondarily by their numbering according to pr:

A task τ has *higher rate-monotonic priority than* τ', written $\tau >_{\mathrm{RM}} \tau'$, iff $\pi_\tau < \pi_{\tau'} \vee (\pi_\tau = \pi_{\tau'} \wedge pr(\tau) < pr(\tau'))$.

The priority relations $>_{\mathrm{FP}}$ and $>_{\mathrm{RM}}$ are total orders on the set of tasks due to the unique numbering given by pr. Therefore, there will never be a non-deterministic choice when scheduling according to the rate-monotonic and fixed-priority disciplines in the system.

The relations $>_{\mathrm{FP}}$ and $>_{\mathrm{RM}}$ are simple since the fixed-priority and rate-monotonic principles are static scheduling principles, i.e. the ordering relations among tasks are independent of the current point in time. For earliest deadline first this is

different as, for a given time point t, the task with the highest priority is that with the shortest distance to its nearest deadline. Hence, the priority relation between tasks may change when some new period starts during the execution.

Let $t \in \mathbb{N}$ and $\tau \in \mathcal{T}$. By $\text{dist}_\tau(t)$ we denote the *distance from t to the nearest deadline (i.e. the start of a new period) of τ*:

$$\text{dist}_\tau(t) = p - t \qquad \text{where} \qquad \left\{ \begin{array}{l} p = o_\tau + n \cdot \pi_\tau \\ \text{for the smallest } n \in \mathbb{N}_+ \text{ so that } p > t \end{array} \right\}$$

Notice that we have a simple setting where the deadline for a task is the same as the start point of its next period. The model is easily extended to cope with deadlines that are earlier than the start of the next period.

A task τ has *higher earliest-deadline-first priority than τ' at time $t \in \mathbb{N}$*, written $\tau >^t_{\text{EDF}} \tau'$, iff.

$$dist_\tau(t) < \text{dist}_{\tau'}(t) \vee (\text{dist}_\tau(t) = \text{dist}_{\tau'}(t) \wedge pr(\tau) < pr(\tau'))$$

The priority relation $>^t_{\text{EDF}}$ is also a total order, and observe that when no task has a deadline between two time points t_1 and t_2, then $>^t_{\text{EDF}} = >^{t'}_{\text{EDF}}$, for every t, t' where $t_1 < t, t' < t_2$. Thus, the earliest-deadline-first priorities can change at time points only when some new period for a task starts.

4.6 Model of Computation

To model the computations of a system, the notion of a state, which is a snapshot of the state of affairs of the individual processing elements, is introduced. Consider a processing element pe_i. For that processing element the state component must record which task in $\mathcal{T}_{pe_i}$ is currently executing, where we denote by $\perp$ that no task is currently executing on pe_i. Furthermore, for every task $\tau \in \mathcal{T}_{pe_i}$, the state component also records the execution time $\epsilon_i(\tau) \in \{bcet_\tau, bcet_\tau + 1, \ldots, wcet_\tau - 1, wcet_\tau\}$ that is needed by τ to finish its job in the current period. We call ϵ_i an *execution vector* for pe_i. Each time a new period for τ starts, a non-deterministic choice is taken concerning the execution time for τ in that period. Thus, a *state component* for pe_i is a tuple (s_i, ϵ_i), where $s_i \in \mathcal{T}_{pe_i} \cup \{\perp\}$ and ϵ_i is an execution vector for pe_i.

4.6.1 System state

A *system state* or just a *state* is an M-tuple $\sigma = ((s_1, \epsilon_1), (s_2, \epsilon_2), \ldots, (s_M, \epsilon_M))$, where (s_i, ϵ_i), is a state component for pe_i, for $i \in \{1, \ldots, M\}$.

A state describes the system in one time unit, and $s_i = \tau$ means that task τ is executing on pe_i for one time unit, while $s_i = \perp$ means that no task is executing on pe_i.

4.6.2 Trace

A *trace* (or *history*) is a finite sequence of states:

$$\hbar = \sigma_1 \sigma_2 \cdots \sigma_k$$

where $k \geq 0$ is the length of $\hbar$, denoted by $\text{length}(\hbar)$. A trace with length k describes a system behavior in the interval $[0, k[$.

Consider a task $\tau \in \mathcal{T}$. The *completed periods of τ in a trace $\hbar$* is the set

$$\text{Completed}_{\hbar}(\tau) \;=\; \{n \in \mathbb{N}_+ \mid o_\tau + n \cdot \pi_\tau \leq \text{length}(\hbar)\}$$

and the *current period number* of τ in $\hbar$ is

$$\text{cpn}_{\hbar}(\tau) = \begin{cases} 0 & \text{if } \text{length}(\hbar) < o_\tau \\ 1 & \text{if } o_\tau \leq \text{length}(\hbar) < o_\tau + \pi_\tau \\ 1 + \max \text{Completed}_{\hbar}(\tau) & \text{otherwise} \end{cases}$$

The n'th *period*, $n \geq 1$, of τ in $\hbar = \sigma_1 \sigma_2 \cdots \sigma_k$ is the (possibly empty) subsequence of $\hbar$:

$$\hbar(\tau, n) = \sigma_{\alpha+1} \cdots \sigma_{\alpha+\pi_\tau}, \text{ where } \alpha = o_\tau + (n-1) \cdot \pi_\tau.$$

Notice that $\hbar(\tau, n)$ consists of π_τ states just when $n \in \text{Completed}_{\hbar}(\tau)$ and fewer (possibly no) states otherwise.

The *execution time* $\text{exec}_\sigma(\tau)$ of a task $\tau \in \mathcal{T}_{pe_i}$, in a state σ is

$$\text{exec}_\sigma(\tau) = \begin{cases} 1 & \text{if } s_i = \tau \\ 0 & \text{otherwise} \end{cases}$$

where $\sigma = (s_1, \epsilon_1), \ldots, (s_i, \epsilon_i) \ldots, (s_M, \epsilon_M))$.

This notion extends to a finite sequence of states as follows:

$$\text{exec}_{\sigma_1 \cdots \sigma_j}(\tau) = \Sigma_{m=1}^{j} \text{exec}_{\sigma_j}(\tau)$$

We denote by $\text{Finished}_{\hbar}(\tau, n) \in \text{Bool}$ whether $\tau \in \mathcal{T}_{pe_i}$ has finished its job in the n'th period, $n \geq 1$, in $\hbar$:

$$\text{Finished}_{\hbar}(\tau, n) = \begin{cases} \text{true} & \text{if } \text{exec}_{\hbar(\tau,n)}(\tau) = \epsilon_i(\tau) \\ \text{false} & \text{otherwise} \end{cases}$$

where ϵ_i is the execution vector for processing element pe_i in the last system state of $\hbar$, or the *zero-execution vector* ϵ_i^0, where $\epsilon_i^0(\tau) = 0$, for $\tau \in \mathcal{T}_{pe_i}$, if the trace $\hbar$ is empty.

4.6.3 Successor states

We shall now define the set of possible successor components states for a processing element pe_i given a trace $\hbar$ with length k. Let ϵ_i be the execution vector for pe_i in the last state in $\hbar$ or the zero-execution vector if the trace is empty. For each $\tau \in \mathcal{T}_{pe_i}$, there are the following choices for the execution vector ϵ' for the next state component of pe_i:

- $\epsilon'(\tau) = 0$, if $k < o_\tau$, i.e. the first period for τ has not started yet,
- $\epsilon'(\tau) \in \{bcet_\tau, bcet_\tau + 1, \ldots, wcet_\tau - 1, wcet_\tau\}$, if $\pi_\tau | \, k - o_\tau$ (π_τ divides $k - o_\tau$), i.e. if a new period for τ starts at time k, then one of the possible execution times for τ is chosen, and
- $\epsilon'(\tau) = \epsilon(\tau)$, if $k \geq o_\tau$ and $\pi_\tau \nmid k - o_\tau$.

Let $EV_{\hbar}(i)$ be the (finite) set of all possible execution vectors for the next state component of pe_i given $\hbar$.

For a given $\epsilon \in EV_{\hbar}(i)$, a task $\tau \in \mathcal{T}_{pe_i}$ is *enabled given* $\hbar$ *and* ϵ, if

- $\epsilon(\tau) > 0$, i.e. $n = \text{cpn}_{\hbar}(\tau) > 0$,
- $\text{Finished}_{\hbar}(\tau, n) = \text{false}$, i.e. τ is not yet finished in the current period, and
- Every task τ' on which τ depends, i.e. $\tau' \prec \tau$, is finished in the n'th period, i.e. $\text{Finished}_{\hbar}(\tau', n) = \text{true}$.

Let $Enable_{\hbar}(\epsilon, i) \subseteq \mathcal{T}_{pe_i}$ be the set of enabled tasks on pe_i given $\hbar$ and ϵ.

The enabled task (if any) with the highest priority is *selected* to execute on the processing element:

$$Select_{\hbar}(\epsilon, i) = \begin{cases} \bot & \text{if } Enable_{\hbar}(\epsilon, i) = \emptyset \\ \tau & \text{if } \tau \text{ is the biggest element in } Enable_{\hbar}(\epsilon, i) \text{ wrt. } >^{k}_{Sch(pe_i)} \end{cases}$$

where $k = \text{length}(\hbar)$ and the priority relations $>_{\text{FP}}$ and $>_{\text{RM}}$ are extended to the time domain in the trivial way: $>^{t}_{\text{FP}} = >_{\text{FP}}$ and $>^{t}_{\text{RM}} = >_{\text{RM}}$, for $t \in \mathbb{N}$.

The set of *possible next component states for* pe_i is defined by:

$$Next_{\hbar}(i) = \{\ (s, \epsilon) \mid \epsilon \in EV_{\hbar}(i) \text{ and } s = Select_{\hbar}(\epsilon, i)\ \}$$

and the set of *possible next system states* is defined by:

$$Next_{\hbar} = \{\ ((s_1, \epsilon_1), \ldots, (s_n, \epsilon_n)) \mid (s_i, \epsilon_i) \in Next_{\hbar}(i), \text{ for } 1 \leq i \leq M\ \}$$

4.6.4 Computation tree

The *computation tree for a system* is a finitely branching, infinite tree, where the root is the empty trace $\langle\,\rangle$ and the internal nodes are (labelled by) system states. Furthermore,

- there is an edge from the root to a distinct node for each system state in $Next_{\langle\,\rangle}$, and
- for every internal node nd in the tree, let $\hbar_{nd}$ be a trace of system states leading from the root to nd. For every system state $\sigma \in Next_{\hbar_{nd}}$ there is an edge from nd to a distinct node labelled by σ.

A *run of the system* is an infinite sequence of states

$$\rho = \sigma_1\, \sigma_2\, \sigma_3\, \cdots$$

occurring on some path starting from the root of the computation tree.

Notice that when the worst and best case execution times are equal for every task in the system, then there is exactly one run of the system, as the scheduling is deterministic.

We call a system *schedulable* if for every run, each task finishes its job in all its periods. We shall now show that this problem is decidable.

4.7 Decidability

We now consider the problem of determining schedulability of a system. We will give an upper bound on the size of the part of the computation tree it suffices to consider when checking for schedulability.

We first establish a periodic behavior of the priority relations among tasks. The two relations $>^t_{\mathrm{FP}}$ and $>^t_{\mathrm{RM}}$ for the static scheduling principles are trivial as they are, in fact, static. We just need to consider the priorities with regard to the earliest-deadline-first principle.

4.7.1 Scheduling situations

To this end, let a *situation at time t*, sit_t, $t \in \mathbb{N}$, be defined by the k-tuple:

$$sit_t = (\mathrm{dist}_{\tau_1}(t), \mathrm{dist}_{\tau_2}(t), \ldots, \mathrm{dist}_{\tau_k}(t))$$

where $k = \mathrm{card}(\mathcal{T})$ and the numbering is given by pr. For a given time t, the earliest-deadline-first priorities can easily be extracted from sit_t (it would actually suffice just to consider situations for tasks that are mapped to processing elements with an earliest-deadline-first discipline).

Consider an arbitrary task τ. It is easy to see that dist_τ satisfies the following properties:

$$\mathrm{dist}_\tau(t) = 1 \iff \mathrm{dist}_\tau(t+1) = \pi_\tau \tag{4.1}$$

$$\mathrm{dist}_\tau(t) > 1 \iff dist_\tau(t+1) = \mathrm{dist}_\tau(t) - 1 \tag{4.2}$$

Therefore, for every $t, c \in \mathbb{N}$:

$$\mathrm{dist}_\tau(o_\tau + t) = \mathrm{dist}_\tau(o_\tau + c \cdot \pi_\tau + t) \tag{4.3}$$

Hence, when the time of the offset for τ has passed, dist_τ becomes periodic with period π_τ and, hence, any multiplum of π_τ is a period as well.

This generalizes to situations in the following way. Let $O_M = \max\{o_{\tau_1}, \ldots, o_{\tau_k}\}$ be the maximal offset in the system, and $\Pi_H = \mathrm{LCM}\ \{\pi_{\tau_1}, \ldots, \pi_{\tau_k}\}$ be the *hyper-period* for the tasks, i.e. the least common multiple of all periods of tasks in the system. Since the period of any task in the system is a divisor of the hyper-period, we have, using (4.3), the following periodic properties:

$$\mathrm{dist}_\tau(O_M + t) = \mathrm{dist}_\tau(O_M + c \cdot \Pi_H + t) \tag{4.4}$$

$$sit_{O_M + t} = sit_{O_M + c \cdot \Pi_H + t} \tag{4.5}$$

for every $t, c \in \mathbb{N}$.

Hence, the infinite sequence of situations

$$Sit = sit_1\ sit_2\ sit_3\ \ldots$$

has, using (4.5), the following form

$$Sit = sit_1\ sit_2\ sit_3\ \ldots\ sit_{O_M}\ (sit_{O_M+1}\ sit_{O_M+2}\ \ldots\ sit_{O_M+\Pi_H})^{\omega} \tag{4.6}$$

4.7.2 Depth of computation tree

Due to 4.6 we meet situations (and scheduling priorities), for any time $t \geq O_M+\Pi_H$, that we have seen earlier. This does not mean, however, that it suffices to consider the computation tree to a depth $O_M + \Pi_H$. The problem is that the execution times may not have "stabilized" yet at time $O_M + \Pi_H$ as the example in Figure 4.3 shows. Even though the utilization $U = 1/3+1/3+2/3 = 4/3 > 1$ and the system obviously is not schedulable, the first deadline is missed three hyper-periods after the maximal offset.

task	execution time $wcet_\tau = bcet_\tau$	period π_τ	offset o_τ
τ_1	1	3	0
τ_2	1	3	1
τ_3	2	3	2

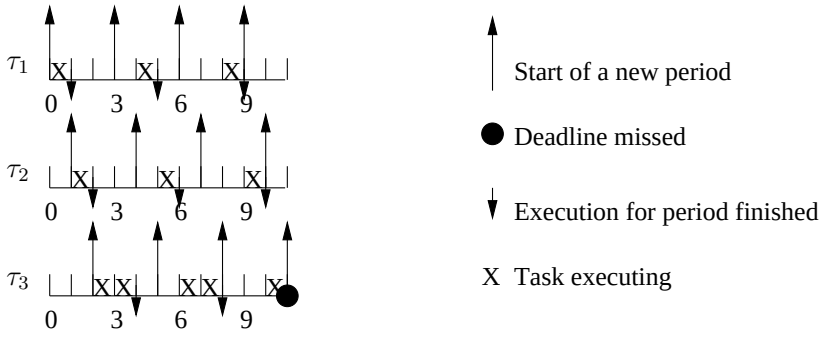


Figure 4.3: Example: deadline missed at time $O_M+3\cdot\Pi_H$, where $O_M = 2, \Pi_H = 3$

Consider a run $\rho = \sigma_1 \sigma_2 \sigma_3 \ldots$ where, for every task, the same execution time is chosen throughout the run, i.e. for $\tau \in \mathcal{T}_{pe_i}$ and $j, k \geq o_\tau$ we have that $\epsilon_i(\tau) = \epsilon'_i(\tau)$, where $\sigma_j = ((s_1, \epsilon_1), \ldots, (s_i, \epsilon_i), \ldots, (s_M, \epsilon_M))$ and $\sigma_k = ((s'_1, \epsilon'_1), \ldots, (s'_i, \epsilon'_i), \ldots, (s'_M, \epsilon'_M))$.

For a given $t \in \mathbb{N}$, let ρ_t be the prefix of ρ up to time t, and $\text{exec}_\rho(\tau, t)$ be the execution time of τ in the current period up to time t in ρ_t, i.e.

$$\rho_t = \sigma_1 \, \sigma_2 \, \cdots \, \sigma_t \quad \text{and} \quad \text{exec}_\rho(\tau, t) = \text{exec}_{\overline{\sigma}}(\tau)$$

where $\overline{\sigma} = \rho_t(\tau, \text{cpn}(\rho_t, \tau))$.

Consider a time $t \geq o_\tau$. At time t, some tasks may not have started yet and, therefore, τ may have been granted more executing time in its current period at time t than one hyper-period later at time $t + \Pi_H$, i.e.

$$\text{exec}_\rho(\tau, t) \geq \text{exec}_\rho(\tau, t + \Pi_H) \geq 0 \tag{4.7}$$

where we exploit that some task is executing on a processing element whenever there is an enabled task on that element – execution time is not wasted.

When a time $t_p \geq O_M$ is reached where for every task $t \in \mathcal{T}$:

$$\text{exec}_\rho(\tau, t_p) = \text{exec}_\rho(\tau, t_p + \Pi_H)$$

then ρ is periodic from time t_p, i.e.

$$\rho = \rho_{t_p} \, (\sigma_{t_p+1} \, \sigma_{t_p+2} \, \ldots \sigma_{t_p+\Pi_H})^\omega$$

An upper bound on t_p is

$$O_M + \Pi_H \cdot (1 + \Sigma_{\tau \in \mathcal{T}} wcet_\tau) \tag{4.8}$$

since, by (4.7), the worst-case scenario is when one task only gets its execution time decreased (by one) when one hyper-period has passed. This upper bound (4.8) can be improved since $\text{exec}_\rho(\tau, O_M) = 0$ for every task τ that starts a new period at time O_M. These tasks satisfy the property: $\pi_\tau \mid (O_M - o_\tau)$. Thus, in order to search the computation checking for schedulability, it suffices to search to the depth: $O_M + \Pi_H \cdot (1 + \Sigma_{\tau \in \mathcal{T}_X} wcet_\tau)$, where $\mathcal{T}_X = \{\tau \in \mathcal{T} \mid \pi_\tau \not| (O_M - o_\tau)\}$. A missed deadline that occurs deeper in the computation tree will also occur at a depth closer to the root, but possibly on another path.

4.7.3 Number of leaves in the computation tree

Let us examine the number of nodes at the following two depths $O_M + \Pi_H$ and $O_M + \Pi_H \cdot (1 + \Sigma_{\tau \in \mathcal{T}_X} wcet_\tau)$ in the computation tree. These numbers are the

minimum and maximum number of paths in the tree that need examination in order to check for schedulability. For a given task τ, we let

- $periodicChoices_\tau$ denote the number of non-deterministic choices for execution time in each period for τ, i.e.

$$periodicChoices_\tau = wcet_{\tau_i} - bcet_{\tau_i} + 1$$

- $initialChoices_\tau$ denote total number of non-deterministic choices for evaluation time of τ until O_M, i.e.

$$initialChoices_\tau = \lceil (O_M - o_\tau)/\pi_\tau \rceil \cdot periodicChoices_\tau$$

- $hyperChoices_\tau$ denote the number of non-deterministic choices for the evaluation time of τ in a hyper-period, i.e.

$$hyperChoices_\tau = (\Pi_H/\pi_{tau}) \cdot periodicChoices_\tau$$

Hence the total number of non-deterministic choices for the system in the time interval $[0, O_M + \Pi_H[$ is

$$totalChoices = \prod_{\tau \in \mathcal{T}} (initialChoices_\tau + hyperChoices_\tau) \tag{4.9}$$

which equals the number of nodes in the computation tree at depth $O_M + \Pi_H$, i.e. the minimum depth in the computation tree to check before a system can be deemed schedulable. Unschedulability may be determined earlier.

The total number of non-deterministic choices for the system in the time interval $[0, O_M + \Pi_H \cdot (1 + \Sigma_{\tau \in \mathcal{T}_X} wcet_\tau)[$ is

$$maxChoices = \prod_{\tau \in \mathcal{T}} (initialChoices_\tau + hyperChoices_\tau \cdot (1 + \Sigma_{\tau \in \mathcal{T}_X} wcet_\tau)) \tag{4.10}$$

which equals the number of nodes in the computation tree at depth $O_M + \Pi_H \cdot (1 + \Sigma_{\tau \in \mathcal{T}_X} wcet_\tau)$, i.e. the depth in the computation tree it suffices to search when checking for schedulability.

4.7.4 A small example showing high complexity

It is easy to give a small system with a huge number of states in the computation tree up to the depth $O_M + \Pi_H$. Consider, for example, the system with three tasks given in Figure 4.4.

task	execution time $(bcet_\tau, wcet_\tau)$	period π_τ	offset o_τ
τ_1	(1, 3)	11	0
τ_2	(1, 4)	8	10
τ_3	(1, 8)	251	27

Figure 4.4: Example: Small example with a huge state space in the non-periodic part

The maximal offset and the hyper-period of this system are:

$$\begin{aligned} O_M &= \max\{0, 10, 27\} = 27 \\ \Pi_H &= \text{LCM}\{11, 8, 251\} = 22088 \end{aligned}$$

and the number of non-deterministic choices for each task in the initial part, each period and each hyper-period are given by:

Task	Periodic choices $periodicChoices_\tau$	Initial choices $initialChoices_\tau$	Hyper-choices $hyperChoices_\tau$
τ_1	3	9	6024
τ_2	4	12	11044
τ_3	8	0	704

Hence the number of nodes at depth $O_M + \Pi_H = 22115$ is

$$totalChoices = (9 + 6024) \cdot (12 + 11044) \cdot (0 + 704) \approx 4.7 \cdot 10^{10}$$

and the number of nodes at depth $O_M + \Pi_H \cdot (1 + \Sigma_{\tau \in \mathcal{T}_X} wcet_\tau) = 176731$ is

$$maxChoices = (9 + 6024 \cdot 8) \cdot (12 + 11044 \cdot 8) \cdot (0 + 704 \cdot 8) \approx 2.4 \cdot 10^{13}$$

4.8 Summary

In this chapter, a formal semantics for MoVES has been given. The concepts of the formalization was informally introduced through an example and thereafter the full formalization was shown. After the formalization of the MoVES model, there was a result regarding decidability of schedulability analysis of systems modelled with MoVES, and a small example showed that the complexity of such analysis can be rather large, especially when periods are well-chosen in order to produce a great hyper-period.

Chapter 5

MoVES Analyses using Timed Automata

Having a decidability result for schedulability analysis of the MoVES computational model, the next question is to get an implementation of the decision algorithm. One way would be to construct a program directly on the basis of the computation tree and the results from Chapter 4 using well-known tree search algorithms such as breadth-first search. Another way would be to reduce the problem to a SAT solving problem, and use sat solvers such as iSAT [37]. However, we will take another approach as we aim at a more general framework supporting both simulation and verification of systems. We will build a model that comprises the components of systems where applications execute on platforms, and exhibit the parallel activities occurring in such systems.

The model will comprise the concepts supported by the ARTS framework as described in Chapter 2, and our approach is to develop a model of systems that can be verified using a model-checking tool. Since the model of computation for systems is discrete, we could use a system like SPIN [35]. But the ARTS framework has notions that are naturally modelled by clocks and timed automata [5]. Therefore, we will use the Uppaal [8, 40] system for modelling, verification and simulation.

We model systems as timed automata composed in parallel - the timing aspects are modelled in the automata modelling tasks, see Section 5.1.2. We assume that

the reader has general knowledge of timed automata and the UPPAAL system, and we will therefore not give introductions to timed automata and the UPPAAL system here. For such introductions we refer to [5] and [8, 40].

In this chapter, we will provide the overview of timed automata models for MoVES. We will explain how the models correspond to the formal model from Chapter 4 and explain the structure of the timed automata models. Not all details will be given here, but in Appendix A the full timed automata templates, together with variable and procedure declarations, will be provided.

In Section 5.1.2, we give four different timed-automata implementations modelling tasks. We aim at implementations that are simply explained and easily understood. The implementations should, however, also provide precise results (i.e. either "schedulable" or "not schedulable"). Efficient analysis is also preferable, so that analyses can be conducted quicker and larger systems can be analyzed. We start at an easily understood implementation where precise results cannot be guaranteed, and end in a much more complex implementation that yields precise results and is efficient, but whose complexity makes it much harder to explain and understand.

This chapter is based on the initial work in [10], work that was later revised in [9] Here, we extend the timed automata models and explain them more in terms of the formal model from Chapter 4. Also, we consider ways to remove inherent non-determinism (see Section 5.3.1), something that has not been considered in previous work.

5.1 Modelling MoVES Using Timed Automata

In Chapter 4, modelling of systems was introduced through a rather informal explanation of the concepts involved in Section 4.1. We now use these concepts to explain how modelling using timed automata can capture the model of computation for MoVES.

The model of computation spans a finitely branching, infinite computation tree, where each node represents a system state:

$$\sigma = ((s_1, \epsilon_1), \ldots, (s_M, \epsilon_M))$$

where s_j is the task currently executing on processing element pe_j, and ϵ_j is the current execution vector for the tasks mapped to pe_j for $j \in 1 \ldots M$. See Figure 4.2 for an example of such a computation tree.

In the timed-automata model for the system T_{sys}, this M-tuple is represented using parallel composition of timed automata for each of the M processing elements:

$$T_{sys} = \|_{j=1}^{M} T_{pe_j} \parallel T_{adm}$$

where T_{adm} is a timed automaton administering changes in dynamic scheduling criteria and it implements *sit* and dist from the formal model. The full timed-automata template of T_{adm}, the *administrating template*, for each verification structure can be found in Appendix A.

5.1.1 Timed-automata model for processing elements

The timed-automata model for a processing element T_{pe_j} is a parallel composition of a timed automaton modelling the operating system of the processing element T_{os_j} and the tasks mapped to it i.e. $\tau_{j_1} \ldots \tau_{j_k}$:

$$T_{pe_j} = \|_{i=1}^{k} T_{\tau_{j_i}} \parallel T_{os_j}$$

The timed-automata model for the operating system on the *j'th* processing element T_{os_j} is a parallel composition of a timed automaton for a controller T_{con_j}, which manages the communication between tasks and processing element, and the individual components of the operating system. Here we include a timed-automata model for the synchronizer T_{synch_j} managing task dependencies, as well as a timed-automata model for the scheduler T_{sch_j} granting execution time to the tasks ready to be executed on the processing element:

$$T_{os_j} = T_{con_j} \parallel T_{synch_j} \parallel T_{sch_j}$$

This structure of timed-automata models for systems follows the concepts and terminology found from ARTS and described in Chapter 2. The same structure is supported by the MoVES language defined in Chapter 3. When using a model-checking tool such as UPPAAL, simulation and counter-example traces are given in terms of this structure as well.

The timed automata comprising a system communicate by synchronous communication and shared variables. In Figure 5.1 the channel communication internally on each processing element and between the different processing elements is shown in terms of communication between pe_j and pe_l. The operating system os_j on processing element pe_j is made up of con_j, $synch_j$ and sch_j. The timed automata for these components T_{con_j}, T_{synch_j} and T_{sch_j} share the channels $\texttt{synchronize}_j$ and $\texttt{schedule}_j$, the same holds for os_l. All operating

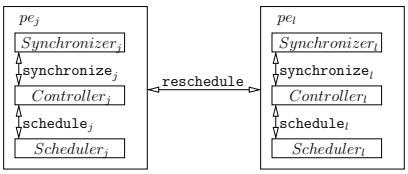


Figure 5.1: Communication internally and between processing elements

systems on the processing elements making up the full system share the channel *reschedule*, which is used to notify other processing elements when a dependency has been resolved. This ensures that $Enable_\hbar(\epsilon, j)$ from the formal model is consistent for all processing elements.

In the formal model it is stated that for a task to be in the set of enabled tasks, every task on which the task in question depends on should be finished in the given period. When a task that has tasks depending on it finishes, we say that the dependency is resolved. If a task is released and there are still tasks on which it depends that have not yet finished in the given period, we say that the task has unresolved dependencies.

We give an overview of the channel communication between processing elements and tasks in Figure 5.2. Suppose that the set of tasks $\{\tau_{j_1}, \ldots, \tau_{j_k}\} = \mathcal{T}_{pe_j}$ is

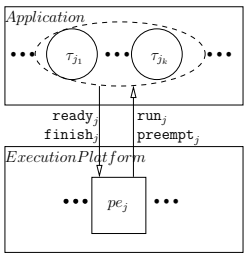


Figure 5.2: Communication between execution platform and application

executed on the processing element pe_j. The timed automata for these k tasks $T_{\tau_{j_1}} \ldots T_{\tau_{j_k}}$ and the processing element T_{pe_j} share four channels ready$_j$, run$_j$, preempt$_j$ and finish$_j$. The model will be constructed so that all channel events will occur at integer time points only, and the correctness of the construction relies on that property.

Before providing timed-automata templates, we give a brief explanation of some of the syntax seen in these automata. Timed automata are generalized finite automata extended with real-valued clocks. Locations in timed automata may be marked with a C (for committed) or U (for urgent). For both types of locations it is the case that time cannot advance in these locations. Furthermore, for committed locations it is the case that a transition leaving the location should be taken immediately, before taking any other transitions in the system.

A double circle indicates the initial location. Names for locations are in dark red text. Transitions can have guards, synchronization events and updates. Guards are in green text, synchronization events are in light blue text and updates are in dark blue text. Guards and updates ending with parenthesis indicate a call to a procedure (code or implementation details for these procedures will not be given here, we instead refer to Appendix A for all implementation details).

A basic timed-automata template for the controller of a processing element can be seen in Figure 5.3. We call it a basic template as some details have been left out. Later in this section, we will give a full timed-automata template for the controller where these details are included. In this basic template, some

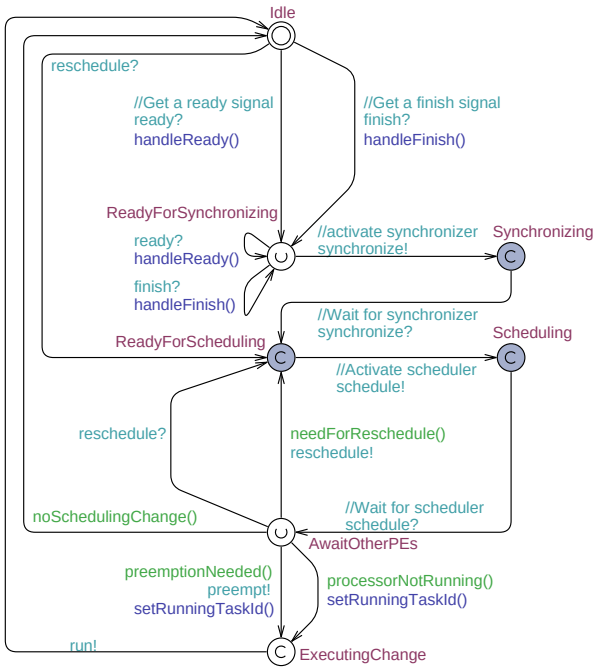


Figure 5.3: Basic timed-automata template for controllers

locations have been shaded while others have not. The shaded (or light blue) locations indicate locations where work internally on the processing element is done (i.e. communication to the synchronizer and scheduler), whereas the unshaded (or white) locations indicate locations where synchronization externally with the tasks is done.

The aim of the controller is to act as a layer between the application and the execution platform. It acts in the following manner:

From the `Idle` location it awaits `ready` or `finish` signals from the tasks mapped to it. After getting the signals from the tasks and acting on them accordingly (through the procedures `handleReady` and `handleFinish`), the synchronizer is activated. When the synchronizer finishes, the controller activates the scheduler and waits for that to finish.

After this, the controller waits for other processing elements to finish their synchronization and scheduling. If synchronization on another processing element resolves a dependency, rescheduling is needed; this is communicated over the channel `reschedule`.

When no more rescheduling is needed, the controller executes the scheduling change if any is needed by use of the channels `preempt` and `run`. Whether or not a scheduling change is needed is determined by the predicates `preemptionNeeded`, `noSchedulingChange` and `processorNotRunning`. Setting the identification of the selected task in the array `tauid` is done by the procedure `setRunningTaskId`. See more on this in Section 5.1.2 where communication between tasks and the platform is explained.

Full timed-automata template for controllers

In Figure 5.4 a full timed-automata template for controllers is given. The template extends the basic template from Figure 5.3. Note that the synchronization channels `ready`, `finish`, `preempt`, `run`, `synchronize` and `schedule` are now arrays indexed with the processing element. This means for example that for pe_1, `ready[1-1]` correspond to the synchronization channel $\texttt{ready}_1$; see Figures 5.2 and 5.1.

We also add an identifier for the processing element `pe`. Notice that identifiers for processing elements are assumed to start at 1 and indexing of arrays start at 0. Therefore, we see indexing of arrays as e.g. `ready[pe-1]` to handle this subtlety.

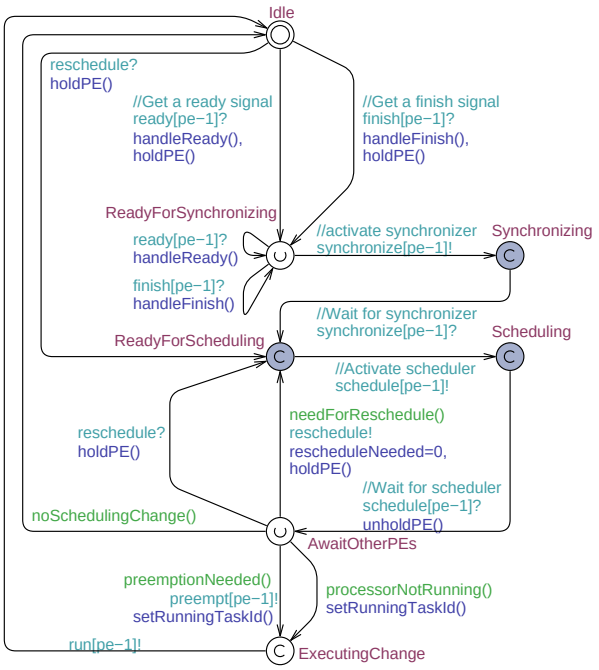


Figure 5.4: Full timed-automaton template for controllers

Furthermore, there is an addition on several transitions of calls to the procedure `holdPE` and on one transition a call to the procedure `unholdPE`. These procedure calls ensure a barrier synchronization of all processing elements in the case that a reschedule is needed.

Timed-automata model for synchronizers

A timed-automata template for the synchronizer of a processing element can be seen in Figure 5.5.

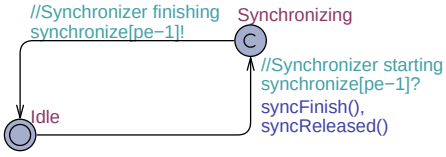


Figure 5.5: Timed-automata template for synchronizers

From the `Idle` location, the synchronizer for pe_j is activated through the channel `synchronize[j-1]` and uses the procedures `syncFinish` and `syncReleased` to maintain $Enable_\hbar(\epsilon, j)$ from the formal model.

The procedure `syncFinish` checks whether the finishing of a task resolves a dependency, in which case $Enable_\hbar(\epsilon, j)$ is updated and a global reschedule may be needed.

The procedure `syncReleased` checks whether a newly released task has unresolved dependencies. If it does not, it is added to $Enable_\hbar(\epsilon, j)$; otherwise suitable data structures keep track of released tasks that are waiting for dependencies to be resolved.

After the synchronization is completed, the synchronizer on pe_j notifies its controller through the channel `synchronize[j-1]`

Timed-automata model for schedulers

A timed-automata template for the scheduler of a processing element can be seen in Figure 5.6.

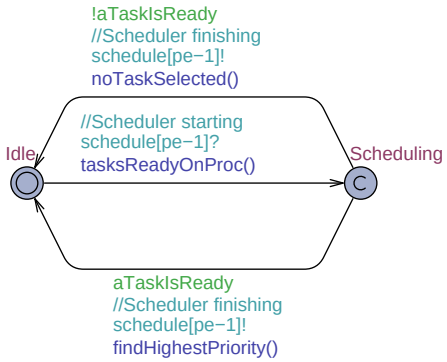


Figure 5.6: Timed automata template for schedulers

When the scheduler on pe_j is activated through the `schedule[j-1]` channel, the procedure `tasksReadyOnProc` makes a check of whether $Enable_\hbar(\epsilon, j)$ is empty and sets the flag `aTaskIsReady` accordingly.

Recall the notion of a system state:

$$\sigma = ((s_1, \epsilon_1), \ldots, (s_M, \epsilon_M))$$

If this flag is not set on the scheduler for pe_j, it corresponds to $s_j = \perp$ in the in the system state, the procedure `noTaskSelected` administers this. If one or more tasks are ready, the procedure `findHighestPriority` finds the biggest element in $Enable_\hbar(\epsilon, j)$ wrt. $>^k_{Sch(pe_j)}$. Note that the actions of the scheduler correspond to $Select_\hbar(\epsilon, j)$ in the formal model.

After completion of scheduling, the scheduler on pe_j notifies its controller through the `schedule[j-1]` channel.

5.1.2 Timed-automata model for tasks

We explain the basics of the timed-automata model for tasks through a very basic timed-automata skeleton for tasks in Figure 5.7 (variable names of the skeleton are commented in Figure 5.8).

In this skeleton, a select statement is added to the syntax of timed automata (as in UPPAAL). This select statement is in yellow color and is explained later in this section. Also, invariants on locations are added, these invariants are in purple color.

We explain the correspondence of this skeleton to the concept of the system state of the formal model. We call this a skeleton as it is not a fully specified timed-automata template, and it serves only to provide an overview of the internal workings of the implementation. In sections 5.1.4, 5.1.6 and 5.1.7 we provide actual timed-automata templates for tasks.

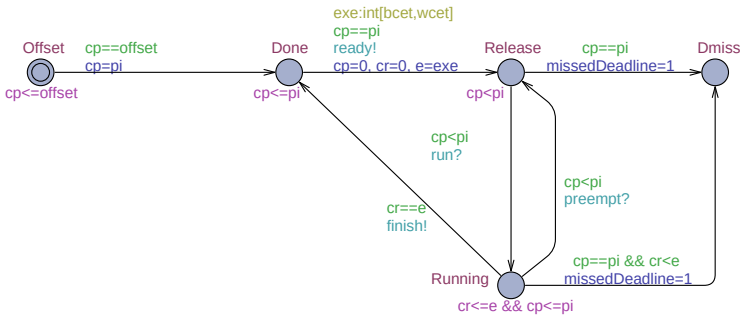


Figure 5.7: Basic timed-automata skeleton for tasks

[1]The clock `cr` should only accumulate when being in the `Running` location

Identifier	Type	Comment
`cp`	clock	Modelling the periodicity of the task
`offset`	int	The initial offset of the task, o
`pi`	int	The period of the task, π
`e, exe`	int	Modelling the non-deterministic choice for execution time
`cr`	clock	Modelling *exec* - the accumulated running time of the task[1]
`missedDeadline`	flag	indicating that the task has missed a deadline

Figure 5.8: Comments for variables in basic task skeleton

Recall the system state:

$$\sigma = ((s_1, \epsilon_1), \ldots, (s_M, \epsilon_M))$$

Each of the s_j for $j \in \{1 \ldots M\}$ correspond to the task in the `Running` location. Note that at most one of the tasks mapped to the processing element pe_j can be in that location at one time instant.

The execution vector $\epsilon_j = \langle \epsilon_j(\tau_{j_1}), \ldots, \epsilon_j(\tau_{j_k}) \rangle$, $j \in \{1 \ldots M\}$ corresponds to the chosen execution times for each of the k tasks mapped to the processing element pe_j at the given time instant. Note that this choice is done on the transition going from the `Idle` location to the `Released` location of the timed-automata template in Figure 5.7. The select statement `exe:int[bcet,wcet]` is syntax for a non-deterministic choice of execution time for the given period and correspond to the $EV_\hbar(i)$ in the formal model.

Note that this choice - together with the selection done by the scheduler - corresponds to $Next_\hbar(i)$ in the formal model. Letting time pass through the use of the clocks `cp` and `cr`, corresponds to a run of the system $\rho = \sigma_1 \, \sigma_2 \, \sigma_3 \, \cdots$ in the formal model.

On the transition going from the `Idle` location to the `Released` location we can observe the guard `cp==pi` and the update `cp=0`. The guard and update handle the periodicity of the task, but they also ensure that `ready` signals are issued at the times dictated by $sit_1, sit_2, \ldots$, formalized in Section 4.7.

Adding communication with the platform

In Figure 5.9 the timed-automata skeleton for tasks has been extended so that it includes communication with the execution platform, i.e. the controller. Note

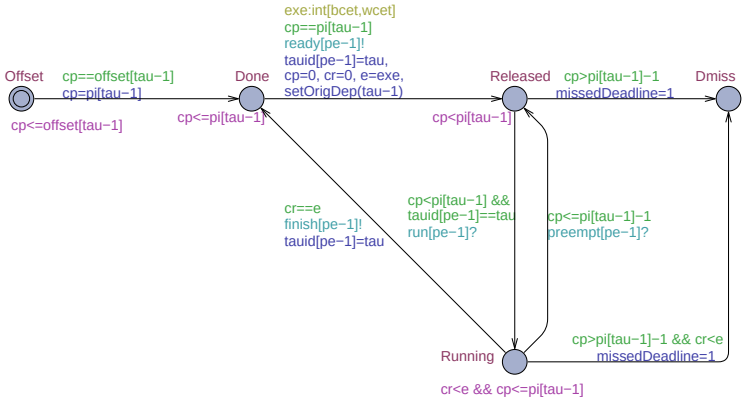


Figure 5.9: Timed-automata skeleton for tasks with platform communication

that `offset` and `pi` are now arrays indexed with the task in question, and the synchronization channels `ready`, `run`, `preempt` and `finish` are indexed with the processing element that the task is mapped to. As an example we say that the channel `ready[1-1]` is the ready channel $ready_1$ for the processing element pe_1, see Figure 5.2. In Figure 5.10 newly added and modified variable names are commented. Note that identifiers for tasks and processing elements `tau` and `pe`

Identifier	Type	Comment
`tau`	int	Identification of the task, τ
`pe`	int	Identification of the processing element that the task is mapped to, pe
`offset`	int array	The initial offset of tasks, o
`pi`	int array	The period of tasks, π
`tauid`	int array	Communicates the identifier of the task to the platform
`setOrigDep`	procedure	Checks if the task should initially be in $Enable_\hbar(\epsilon, pe)$, see Section 5.1.1

Figure 5.10: Comments for variables in task skeleton with platform communication

are assumed to start at 1; the array placements start at 0. Therefore, to handle this subtlety, the indexing of arrays are `tau-1` and `pe-1`.

Ordering *ready* and *finish* signals

One major decision to be made when making the timed-automata implementation is how to order signals sent from different tasks to the platform. If no ordering is done, several temporary "invalid" scheduling decisions may be made, i.e. a task may be selected to execute only to be preempted in zero time if a dependent task's *finish* signal is handled later.

The ordering can be conducted in several ways. Here are three different strategies for ordering:

1. No ordering used. This can result in several temporary "invalid" scheduling decisions to be made, and in quite inefficient schedulability analysis. We will not pursue this strategy further here.

2. Any *finish* signals on a processing element are handled before *ready* signals. This means that a correct local scheduling decision will be made. There is, however, still a chance that finishing tasks on other processing elements will require a reschedule, as one processing element will go through the steps of synchronization and scheduling before the next starts its steps. Note that in order to handle *finish* signals first, knowledge of these must be available before that time instant.

   We use this strategy in the stop-watch automata implementation in Section 5.1.4 and in the implementation where the running time is discretized in Section 5.1.6.

3. All *ready* and *finish* signals are accepted before any steps of synchronization or scheduling is done. In this way, handling *finish* signals before *ready* signals can be done without having knowledge of which signals will be coming before the time instant where they actually do.

   We use this strategy in the alternate stop-watch automata implementation in Section 5.1.5 and in the implementation without clocks in Section 5.1.7.

5.1.3 Preemption of tasks and different timed-automata implementations

As noted in Figure 5.8 by the clock `cr`, this clock should only accumulate when the task is in the `Running` location. If a task is temporarily preempted, the accumulating clock should be stopped. Usual operations on clocks are to reset their values to a new specific value. If we wish to stop a clock temporarily and

start it again later, the automata instead becomes a more general hybrid form. Here, we call a timed automaton with at least one clock that can be stopped a stop-watch automaton.

In [4], Alur et al. proves that, in general, the reachability problem for stop-watch automata is undecidable but also that certain subsets of stop-watch automata problems are decidable. We will elaborate more on this in Section 5.4. In order to create an implementation for the decision algorithm for schedulability analysis as stated in Chapter 4, we give four different automata models for tasks, each of these handle the problem of preemption in a slightly different way:

1. In Section 5.1.4, we start with a stop-watch-automata implementation that is easily explainable and understandable. We use the syntax of an experimental version of UPPAAL that includes stop watches. The analysis of this model, however, relies on the reachability of models including stop watches. This problem, in general, is undecidable. More on this in Section 5.4.

2. Then, in Section 5.1.5 we provide an implementation that still uses stop watches, but this model lies within a decidable subset of problems on stop-watch automata, again see Section 5.4 for more details. However, it is not clear whether the developmental version of UPPAAL that includes the use of stop watches, which we use to conduct the analysis, provides precise results for this specific type of problem. This uncertainty is due to the fact that an over-approximation is used in the reachability analysis.

3. In Section 5.1.6 we turn to a third implementation, where we stay within the syntax for timed-automata without stop watches. In this version, we discretize the running time of tasks, i.e. `cr` in Figure 5.7 is discretized. Analysis on the basis of this implementation of the task model is not very efficient, especially for tasks with large periods.

4. Therefore, we finally propose in Section 5.1.7 an automata model (without clocks) that has been genuinely discretized, and where only time points with real interest are examined. The clocks `cp` and `cr` are replaced by a counter that finds the next "interesting" time point through a list of task release times that can be generated statically and through knowledge of task finishing times, which are identified dynamically. The interesting time points where tasks can be released $sit_1, sit_2, \ldots$ have already been identified in the construction of the timed automation T_{adm}. This implementation gives precise results and is efficient when it comes to analysis. It is however, not quite as easily explainable and understandable as the three other implementations.

5.1.4 Stop-watch automata model

In Figure 5.11 we provide a timed-automata template for tasks using stop watches. The clock `cr` is a stop watch that is only accumulating in the `Running` and `RunningA` locations. The syntax for stop watches is as follows: The invariant `cr'==1` indicates that the clock is accumulating (i.e. the rate is 1), and `cr'==0` indicates that the clock is stopped (i.e. the rate is 0). Note that an ex-

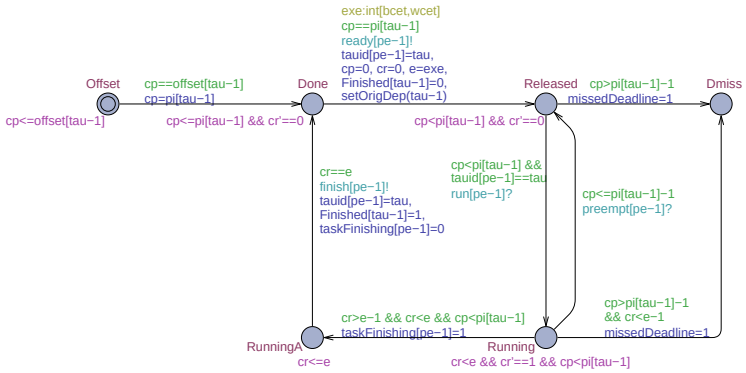


Figure 5.11: Task model using stop-watch automata

tra location `RunningA` is added between the `Running` and `Done` locations. This is done to handle the issue of acting on *finish* signals before *ready* signals as explained previously. A flag for each task in the array `taskFinishing` is set when entering this extra location, and the invariant of `Running` is altered to ensure that the location is left before `cr` reaches the value of `e`. In this way, the system has knowledge of all *finish* signals before they should be acted on.

The timed-automata templates for the stop-watch automata model, including the task template as well as all other templates (e.g. controller, synchronizer, scheduler and administrating template), can be found in Appendix A.1.

5.1.5 Alternative stop-watch automata model

Figure 5.12 shows an alternative implementation using stop watches. In this timed-automata template, we add extra "zero-time" locations to all the locations where *ready* or *finish* signals can be issued. By zero-time we mean locations where clocks cannot advance. These locations work as polling locations. Whenever the platform receives either type of signal, it polls all other tasks to ensure that all signals are being handled. A slight change in the controller is needed

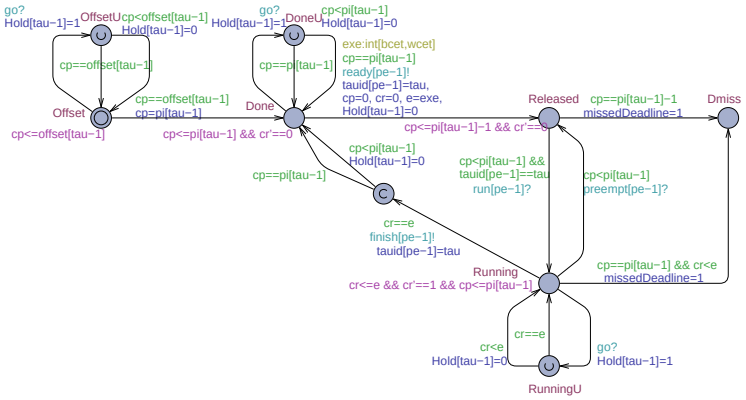


Figure 5.12: Alternative task model using stop-watch automata

in order to follow this strategy, but nothing major. There is a flag in the array `Hold` for each task that ensures that all tasks are being polled.

The timed-automata templates for the alternative stop-watch automata model, including the task template as well as all other templates (e.g. controller, synchronizer, scheduler and administrating template), can be found in Appendix A.2.

5.1.6 Model with discretization of the running time

In Figure 5.13 we provide a timed-automata template for tasks where the running time has been made discrete.

In this model, `cr` is an accumulating variable. A clock `x` is added to handle the discretization. Each time unit `x` ensures that `cr` is increased by one. There is a flag in the array `disc` for each task that ensures that `cr` for all executing tasks is being increased each time unit. Finally, the array `taskFinishing` contains a flag for each task to ensure that the system has knowledge of all *finish* signals before they should be acted on.

The timed-automata templates for the model with discretization of the running time, including the task template as well as all other templates (e.g. controller, synchronizer, scheduler and administrating template), can be found in Appendix A.3.

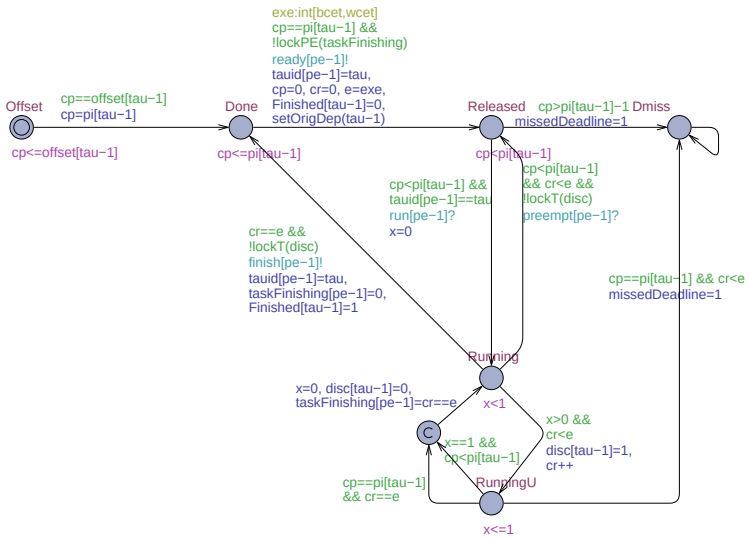


Figure 5.13: Task model using discretization of the running time

5.1.7 Genuine discrete model (no clocks)

In Figure 5.14, we show a timed-automata template for tasks, which is a genuine discrete model. In this implementation, all clocks are eliminated and a counter is instead introduced to manage the timely aspects of the model. We will not explain this model, as the previous three models have been explained from the timed-automata skeleton for tasks. We will instead give explanations that relate this model directly to the formal model defined in Chapter 4 and just give the general idea behind the model.

The general idea is to replace all clocks in the system with a counter. This counter is placed in the timed automaton administering changes in dynamic scheduling criteria, T_{adm}. The reason for placing it here is that the times for *sit* in the formal model are exactly the times when tasks are being released. At any time after some scheduling has finished, T_{adm} determines whether the next interesting time point is the next task release (given by *sit*), or a task finishing, which for each task is updated in an auxiliary variable `cfin` that is maintained by the procedures `preemptUpdate`, `runUpdate` and `finishUpdate`.

In the task template shown on Figure 5.14, the three locations `Done`, `Released` and `Running` can still be identified. There is an extra location, `Dpoll`, which acts as a polling mechanism for whether or not the specific task is being released.

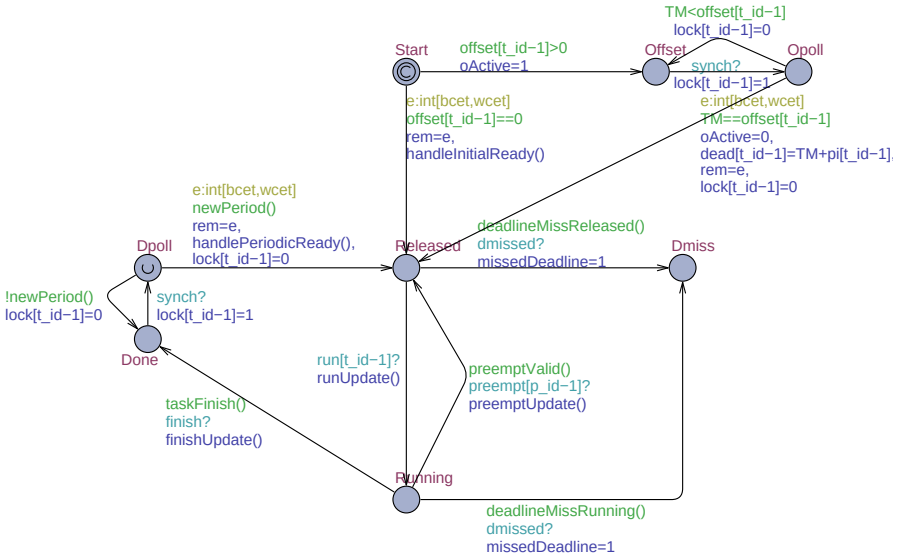


Figure 5.14: Task model with no clocks

This is checked with the procedure `newPeriod` that effectively replaces the `ready` signal from the previously presented templates.

A few optimizations have been made to make the model more efficient in a verification context. Observe that the `finish` signal now originates from T_{adm} rather than the task automaton. Also, after the offset (i.e. the locations `Start`, `Offset` and `Opoll`), the task moves directly to the `Released` location rather than `Done`.

The timed-automata templates for the genuine discrete model, including the task template and all other templates (e.g. controller, synchronizer, scheduler and administrating template), can be found in Appendix A.4.

5.2 Non-Determinism in MoVES vs. Timed-Automata Models

In Section 4.6.4, we formalized the notion of a schedulable system to be that for every run of the system, starting from the root of the computation tree, each task finishes its job in all its periods. The computation tree is finitely branching,

and each branch corresponds to a non-deterministic choice of execution time for some task at some time point, i.e. in the formal model, all non-determinism is due to choices of tasks' execution times.

In the timed-automata models given in this chapter, however, this is not the case. All non-determinism from the branching of the computation tree in the formal model is preserved, but when modelling using networks timed automata composed in parallel, inherent non-determinism occur. If nothing else is defined, whenever two or more timed automata that are composed in parallel can fire a transition, there will be a non-deterministic choice of which of these transitions to take (first). This introduces a lot more non-determinism into the model than what is contained in the formal model.

When aiming for a model that can be automatically verified efficiently, reduction of this inherent non-determinism is desirable. However, any extra functionality added to the networks of timed automata will make them so complex that they become very hard to understand. If the network of timed automata has an ordering; each time two or more automata can fire a transition, the ordering decides which automata is allowed then the inherent non-determinism can be removed, and the only non-determinism left will be that from choices of execution time as in the formal model.

5.3 Analyses using Timed-Automata Models and Uppaal

With the timed-automata models provided in this chapter, schedulability analysis of MoVES systems using `Uppaal` can be conducted. The aim of the schedulability analysis is to detect whether any deadlines of the system are missed. Since the timed-automata templates for tasks all have a flag `missedDeadline` that is set whenever a deadline is missed, schedulability analysis is checking for reachability of a state where this flag is set.

UPPAAL uses a subset of *Timed Computation Tree Logic (TCTL)* where modalities (path quantifiers `E` and A, state-quantifiers `[]` and `<>`) are only allowed at the outermost level. This gives four combinations:

`E<>` ϕ**:** There exists a path, where there exists a state where ϕ holds

`A[]` ϕ**:** On all paths, in all states ϕ holds

`E[]` ϕ**:** There exists a path, where in all states ϕ holds

`A<>` ϕ**:** On all paths, there exists a state where ϕ holds

The schedulability analysis can be conducted with the following query:

```
A[] !missedDeadline
```

which reads: On all Paths, in all states, the flag `missedDeadline` is never set.

If this query is satisfied, no deadlines of the system are ever missed. If however this query is not satisfied, the UPPAAL system can generate a counter example in the form of a trace of a possible way of reaching a missed deadline is given. This can be examined in the UPPAAL simulator, or used to generate a diagram, which is what we do in the tool explained in Chapter 6 that can generate *MoVES traces*.

For the model where the running time is discretized, and the model without clocks verification can be conducted using the official version of UPPAAL. However, for the two models with stop watches, one must use a developmental version of UPPAAL, where stop watches are allowed. For the decision procedure for reachability in this version of UPPAAL, an over-approximation is used.

It is worth noting here, that in order to conduct efficient verification of timed-automata models, one needs to limit the search space as much as possible. It is important to bound all integer variables, and also consistent use of constants whenever possible can help reducing the state space. One major trick in reducing the state space for timed-automata models is to remove non-determinism introduced when composing timed-automata in parallel, we will elaborate on this in Section 5.3.1.

5.3.1 Reduction of non-determinism in Uppaal models

In the UPPAAL syntax for instantiating networks of timed automata, there is a construction that allows for ordering of individual instantiated automata called *Priorities on processes*. This ordering does exactly what is described in Section 5.2: Whenever two or more automata can fire a transition, the one with the higher priority is selected. This means that the inherent non-determinism introduced by modelling using timed automata composed in parallel can be removed using this ordering. However, when using *Priorities on processes*, UPPAAL is unable to generate a counter example trace. Still, the result of the schedulability analysis can be found (i.e. *Property is satisfied* or *Property is NOT satisfied*).

5.4 Summary

In this chapter we have shown how the formal model of MoVES presented in Chapter 4 can be implemented using timed automata. We have provided explanations of how the formal model and the notion of a system state relates to the timed-automata composed in parallel, and we show how the timed-automata models follow the structure of ARTS as explained in Chapter 2. Finally, we provided timed-automata templates for all the components comprising a system in MoVES, explanations of the most important design decisions and how the timed automata implement the different aspects of the formal model.

A major aspect in creating a timed-automata implementation is the decision of how to deal with preemption. We already referred to Alur et al. [4] on how the general reachability problem for stop-watch automata is undecidable. In [4], however, there is a decidability result showing that if guards and invariants for stop watches are limited to the form $c \leq v$ and $c \geq v$, where c is a clock and v is an integer (i.e. no strict equalities), these can be encoded as normal timed automata, and the reachability problem for this subset is therefore decidable. It seems that especially the alternative stop-watch model presented in Section 5.1.5 can be constructed to follow these limitations. This indicates that this problem lies within the decidable subset of problems for stop-watch automata. Whether the algorithm used in the developmental version of Uppaal where stop watches are allowed uses the over-approximation or gives exact results for these problems is still unclear.

Chapter 6

The MoVES Tool

In this chapter we present the tool MoVES, which is a framework for modelling and verifying embedded systems. It is developed to assist in the early phases of embedded systems design.

The MoVES tool can be used to conduct schedulability analysis and has the potential to reason about different types of resource usage such as memory usage and power consumption. The framework has a modular, parameterized structure supporting easy extension and adaptation of the MoVES language as well as choice of verification backend.

First, in Section 6.1, we show how the MoVES tool can be used from a user's perspective, give brief explanations on how to conduct schedulability analysis using the MoVES tool, and show the use of some options that can be used for different types of analyses.

In Section 6.2, we explain the structure of the framework that makes up the MoVES tool. This explanation is done to give advanced MoVES users and developers an idea of how parts of the MoVES tool can be modified, e.g. if wanting to use other verification backends with the MoVES tool. The entire process is highlighted: from a MoVES specification and *verification structure* through an auto-generated *verifiable implementation* to be used for the verification. All this results in a verification result and possibly a trace showing a counter example

of the verification. Not all implementation details are given here, but the full source code for the MoVES tool can be found in Appendix B.

Although the tool has been developed to be independent of verification backend, the current development is based on timed-automata models and use the UPPAAL model checker as backend. Therefore, the following explanations of the use and structure of the MoVES tool will given on the basis of this. In the summary, Section 6.5, we discuss the use of different verification backends.

This chapter is based on the initial work first examined in [45] and later further substantiated in [12]. We extend the work with more explanations of usage of the MoVES tool and more in-depth explanations of its structure.

6.1 A User's Perspective of the MoVES Tool

The user specifies a system using the MoVES language as presented in Chapter 3. A specific verification structure (e.g. the implementing timed-automata models described in Chapter 5) can be used as a basis for the analysis conducted by the MoVES tool. A verifiable implementation (e.g. a timed automaton) of the system is constructed on the basis of the formal model from Chapter 4 and automatic verification of the system can be conducted.

6.1.1 Default analysis using the MoVES tool

The simplest way to use the MoVES tool is to use the default settings. Currently, the default settings use the genuine discrete verification structure that is provided in Section 5.1.7.

Consider the windmill control system and verification property shown in Section 1.2.1 specified in a file with the file name `windmill.mvs` using the syntax of the MoVES language specified in Chapter 3. Verification using the MoVES tool with default settings can be done with the following command:

```
moves windmill.mvs
```

The result of such verification is either *Property is satisfied* or *Property is NOT satisfied*. In the case of the latter, a trace showing a counter example of the verification is also generated.

In the example of the windmill control system, verification of schedulability property is not satisfied, e.g. schedulability analysis of deadline. The verification result from MoVES looks like this:

```
Verifying property 1 at line 1
 -- Property is NOT satisfied.
Showing counter example.
      | 0 2 4 6 8 0 2 4 6 8 0 2 4 6 8 0 2 4 6 8 0 2 4 6
T1    | ++  ++  ++  ++  ++  ++  ++  ++  ++  ++  ++  ++  ++
T2    | OO+   +     OO+   +     OO+   +     OO+   +
T3    | OOOO++OO++  OOOO++OO++  OOOO++OO++  OOOO++OO++
T4    |                                         OO++OOX
T2_T3 | OOO+  O+    OOO+  O+    OOO+  O+    OOO+  O+
```

We call this type of trace a *MoVES trace*. In a MoVES trace, the symbols indicate when tasks are: a) executing (`+`), b) released with execution time remaining (`O`), c) finished executing or without execution time remaining ( ) and d) missing deadlines (`X`). Table 6.1 shows how these *task states* correspond to the locations in the stop-watch automata from Figure 5.11. Also, the corresponding symbol used in a MoVES trace is given for each of these task states.

Table 6.1: MoVES trace symbols and corresponding locations

	Task state	Automata location(s)	Symbol
a	executing	`Running`	`+`
b	released	`Released`	`O`
c	done	`Done`,`Offset`	
d	missed deadline	`Dmiss`	`X`

An example of verification of a satisfied property using the MoVES tool looks like this:

```
Verifying property 1 at line 1
 -- Property is satisfied.
```

6.1.2 Analysis with a specific verification structure

Certain systems or types of verification may be better suited to using different verification structures. If a user wishes to conduct an analysis using a specific verification structure, this can be done using one of the options provided for the MoVES tool:

-sw Using the verification structure using stopwatches from Section 5.1.5

-drt Using the verification structure where the running time is discretized from Section 5.1.6

-nc Using the genuine discrete verification structure from Section 5.1.7

Each of these verification structures is supported by specific verification backend. The verification structure specified by the **-sw** option is supported by a developmental version of UPPAAL, where stop watches are included, as verification backend. The verification structures specified by the other options are supported by the official version of UPPAAL as verification backend.

Note that only one of the stop-watch verification structures has been implemented in the MoVES tool. There is no problem in including the stop-watch verification structure from Section 5.1.4 in the MoVES tool also, it only requires including it in the batch scripts presented in Section 6.3.

If the user has specified a system and specification property in a file with the file name *example.mvs*, this can be verified on the basis of the verification structure using stop watches from Section 5.1.5. Using the MoVES tool, the verification is conducted with the following command:

```
moves -sw example.mvs
```

This verification uses the developmental version of UPPAAL, where stop watches are included, as backend. As verification using this version is based on an over-approximation, an example of a verification could be:

```
Verifying property 1 at line 1
 -- Property MAY NOT be satisfied.
Showing counter example.
      | 0 2 4 6
T1    | ++    +
T2    | OOO+++O
T3    |    OOOX
T1_T2 | OO+   O
```

Note the `Property MAY NOT be satisfied` result, indicating that the verification uses the aforementioned over-approximation.

Reducing non-determinism leading to no trace generation

As explained in Section 5.3.1, the UPPAAL syntax allows for reduction of the inherent non-determinism introduced by modelling systems through timed automata composed in parallel. The user can enable this reduction of non-determinism by following any of the available options by **-nt**. For example, using the verification structure where the running time is discretized from Section 5.1.6 as a basis for verification can be done as follows: A system and verification property is specified in a file with file name *example.mvs*. The following command conducts the verification using the MoVES tool:

```
moves -drt-nt example.mvs
```

A possible result of this verification could be the following:

```
Verifying property 1 at line 1
 -- Property is NOT satisfied.
```

Note that even though the verification property is not satisfied, no counter-example trace is generated, because UPPAAL is unable to generate a counter example trace when using *Priorities on processes* as explained in Section 5.3.1.

So far, we explain the MoVES tool to the extent shown in Figure 6.1.

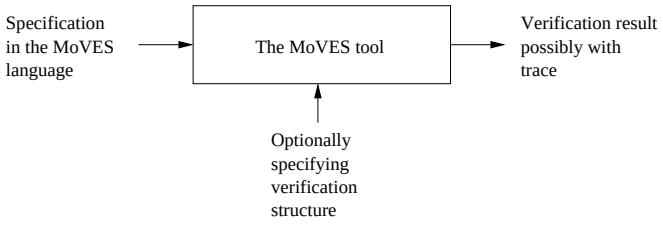


Figure 6.1: The MoVES tool from a user's perspective

In the following section, we present the parts that make up the MoVES tool.

6.2 The MoVES Framework

We call the MoVES tool a framework, as it is a flexible and extendable tool. It provides a generic form of analysis that can be made on the basis of concrete verification structures using concrete verification backends.

The basic idea behind developing the MoVES tool as a framework is that the user can easily choose the verification structure that is best suited for the desired verification. Generally, the wish is to have a flexible and extendable tool, in which slight chances in the verification structure do not require its recompilation.

In developing the formal model in Chapter 4 and the timed automata implementations in Chapter 5, the framework-structure of the MoVES tool has proven very useful. It makes it easy to experiment with small alterations to implementations and verification structures, as no recompilation of the tool is needed for small chances. Furthermore, quick verification of a set of standard examples can easily be conducted when a new or altered verification structure is tested.

The MoVES tool has been developed in the functional programming language SML [51]. No further introduction to functional programming or SML is given here. We refer to *Introduction to Programming using SML* [28] and the website for Moscow ML [55] for further information. The full concrete SML implementation of the MoVES tool can be found in Appendix B. The purpose of this chapter is to convey the main idea only.

Generally, the MoVES tool can be explained in four parts:

Frontend: The system specification and the desired verification property are parsed and represented in suitable data structures. We call these data structures a *MoVES syntax tree*.

Model generation: On the basis of the MoVES syntax tree and the chosen verification structure, the MoVES tool constructs a verifiable implementation. This implementation is suitable for verification using a specific backend.

Verification: The verifiable implementation is verified by the backend that matches the chosen verification structure. A verification result is then found possibly together with a counter-example trace.

Trace generation: Because the result of the verification is given in terms of the verification backend and the structure used, the trace generator translates any possible counter-example trace to a diagram consisting of time points,

and what tasks are executing at these time points. The diagram can be understood directly based on the specification in the MoVES language.

In the following, we will present these four parts of the MoVES tool in more detail. Note that this presentation is based on how this has been done in terms of verification structures given in the UPPAAL syntax. This is so that the general idea of the MoVES tool can be understood.

6.2.1 Frontend

In Figure 6.2, the idea of the frontend of the MoVES tool is shown.

Figure 6.2: The frontend for the MoVES tool

In order to capture the information from the system specification, the specification is parsed and represented in suitable data structures: a MoVES syntax tree. The parsing of the system specification and generation of the MoVES syntax tree have been implemented using the syntax for the Moscow ML lexer generator (mosmllex) and parser generator (mosmlyac) (see [55] for further information on the syntax). The full lexer and parser specifications can be found in Appendix B.1. Here is also the abstract syntax used for the frontend and model generator, as well as a few auxiliary functions used for parsing.

In Section 4.2, a few definitions describing well-formed parts of the system, e.g. same period for all tasks in the same application. These definitions should be included in the frontend, but has not been at this point.

6.2.2 Model generator

In Figure 6.3, the idea of the model generator of the MoVES tool is shown.

Model generation by the MoVES tool is basically the action of adding information in terms of attributes of the specified system to the desired verification structure. At the current time, all verification structures that users can select between are given in the UPPAAL syntax. A verification structure is basically an annotated XML representation of a network of UPPAAL timed automata.

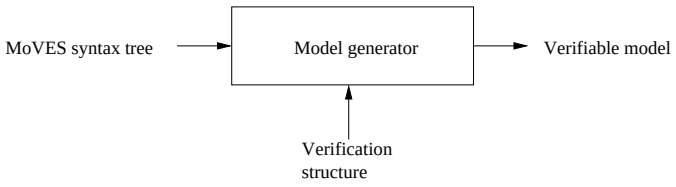


Figure 6.3: The model generator

The first step in adding information to a verification structure given in the UPPAAL syntax is to identify the areas that are system dependent and those that are system independent. We denote areas of the verification structure that are the same no matter what system is modelled as *system independent*. These areas are just copied directly into the resulting model. The areas of the verification structure that differ from system to system, such as number of processing elements and tasks, scheduling principles, tasks' timely properties etc., we call *system dependent*. These are altered to make the implementing model reflect the specified system.

For the model generator in the MoVES tool, we have chosen to insert tokens in the verification structure to identify these parts. These tokens are as follows:

//System-Dependent Decl	Declarations of constants, variables and procedures that are dependent on the actual modelled system. These are things like the number of processing elements and tasks, scheduling principles chosen for the specific processing elements and timely properties for individual tasks such as periods and offsets.
//System-Independent Decl	Declarations of constants, variables, procedures and individual timed-automata templates that are independent of the system modelled. These are the general data structures and procedures that are in place. They are the same, no matter which system is modelled.
//System-Dependent Inst	Instantiations of the network of timed automata that are dependent on the modelled system. Usually all instantiations are system dependent, as the number of tasks and processing elements and some of their properties are specified here. The differentiation between system-dependent and -independent instantiations is made to ensure that further development can be conducted in the same setting.
//System-Independent Inst	Instantiations of the actual timed automata that are not dependent on the actual system modelled. This is usually empty; see also system-dependent instantiations above.

The tokens indicate the start of system-dependent and -independent declarations of variables and procedures as well as system-dependent and -independent instantiations. All system-dependent declarations are assumed to be in the global declarations and not in the local declarations of the individual timed-automata templates.

The following is an example of the beginning of the global declarations for a verification structure specified in the UPPAAL syntax. Note that in Appendix A, the full specification of the four different verification structures can be found, where these tokens are included.

```
clock TM;
const int FP = 0, RM = 1, EDF = 2; //symbolic representation
                                   //of scheduling principles
//System-Dependent Decl
```

```
const int M = 2;
const int N = 3;
const int MN = 3;

const int[FP,EDF] processorScheduling[M] = FP, RM;

                                  ⋮

const int MaxExe=6;
const int MaxPi=30;

//System-Independent Decl
//Synchronization channels
broadcast chan reschedule; //broadcast channel for rescheldulingg
                           //after a task has finished
chan synchronize[M], schedule[M];

                                  ⋮
```

Adding most of the information is straightforward, such as the number of processing elements and tasks that can just be counted. Also, directly-specified information such as scheduling principles of processing elements and timely properties for tasks, e.g. periods and offsets, can be found directly from the specification.

A collection of SML functions have been developed in order to extract this information from the MoVES syntax tree and add it to the XML representation for the network of timed automata given by the verification structure. We will not explain all of these functions here, only highlight the most important. See Appendix B.2 for the concrete SML implementation of all of these functions.

Scheduling situations

An important construction in the model generation done by the MoVES tool is generating time points for scheduling situations (see *sit* in the formal model in Chapter 4). This is done by the function *mkBothULists* with the following SML signature:

```
taskDescription = offset * period * taskid
```

```
order = taskid list
optimize = "U" | "O"
taskReleases = sit list
prioritizedReleases = taskReleases * priority list

mkBothULists = taskDescription list -> order -> optimize ->
                  prioritizedReleases * prioritizedReleases
```

This function takes a list of task descriptions as argument, one for each task in the system is in the list. A task description is a triple (o, p, e), where o is the task's offset, p is the task's period and e is the external representation of the task or the `taskid`. The function generates two lists of scheduling situations (see *sit* in Chapter 4) or `taskReleases`, as well as a list of priorities for each of these situations to be used when earliest-deadline-first scheduling is specified. We call a list of scheduling situations together with a list of priorities `prioritizedReleases`. The first of these corresponds to scheduling situations occurring before the maximal offset has been reached, whereas the second corresponds to those occurring after the maximal offset has been reached, at which point the situations are periodic.

Let us take an example to highlight the use of `mkBothULists`. Consider a system consisting of three tasks, T1, T2 and T3 with the following properties:

Task	Period (π)	Offset (o)
T1	3	0
T2	3	1
T3	3	2

The following value `tdl` represents the list of task descriptions for this system in SML syntax:

```
val tdl = [(0,3,"T1"),(1,3,"T2"),(2,3,"T3")]
```

Applying the function `mkBothULists` to the list of task descriptions, where the order of the external representation of tasks is T1,T2,T3 and with unoptimized list generation, can be done as follows:

```
mkBothULists tdl ["T1","T2","T3"] "U"
```

This function application generates the following pair of prioritized releases:

```
( ([1,2],[ [1,2,3], [1,2,3] ]),
  ([3,4,5],[ [1,2,3], [3,1,2], [2,3,1] ])
)
```

We see that there is one scheduling situation at time point 1 before the maximal offset, 2 is the maximal offset. Initially and throughout the time until the maximal offset, the priority lists show that task prioritization according to earliest-deadline-first is such that the first task (in terms of the order of external representations) has highest priority and the third has lowest priority, i.e. T1 has highest priority and T3 has lowest priority.

After the maximal offset, there are scheduling situations at time points 3, 4 and 5 (after time point 5 the scheduling situations and prioritized releases become periodic). Before the scheduling situation at time point 3, the priority list still gives T1 first priority, T2 second priority and T3 third priority. At time point 3, the priority list shows that T1 has third priority, T2 has first priority and T3 has second priority. At time point 4, the priority list shows that T1 has second priority, T2 has third priority and T3 has first priority. This means that at time point 5, the scheduling situations become periodic and acts as time point 2, and the first element in the prioritized releases is once again valid.

We can generate prioritized releases with optimized lists, so that consecutively following scheduling situations that have the same priority list will be omitted, i.e. only changes in priority lists will be included. Applying `mkBothULists` to the system with the same order on external representation of task but with optimized lists can be done as follows:

```
mkBothULists tdl ["T1","T2","T3"] "0"
```

This function application generates the following pair of prioritized releases:

```
( ([2],[ [1,2,3] ]),
  ([3,4,5],[ [1,2,3], [3,1,2], [2,3,1] ])
)
```

There is no semantical difference, just that the repetition of the priority list `[1,2,3]` at time point 1 is removed. The optimized lists can easily be used as long as the scheduling situations are only used to manage earliest-deadline-first priorities. But when using the scheduling situations to capture interesting time

points as with the genuine discrete model, see Section 5.1.7, all time points must be preserved. In that case, unoptimized lists must be used.

The full source code for the model generator written in SML can be found in Appendix B.2. This source code includes all functions that are used to create the model generator.

6.2.3 Verification of properties

In Figure 6.4, the basic idea of verification with the MoVES tool is shown.

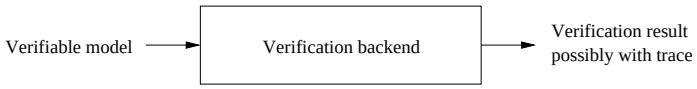


Figure 6.4: Verification in the MoVES tool

The basic idea of verification is to just let the verification happen. The verifiable implementation is generated for a specific verification backend. As the verification backend can be a model checker developed externally (in which we have no control), we just let the backend conduct the verification and thereby get a verification result with possible counter-example trace.

In the case of the verification structures presented here, we have shown the use of two different verification backends: 1) the original UPPAAL model checker, and 2) the developmental version of UPPAAL including the use of stop watches. If different verification backends are introduced into the MoVES tool, verification structures, model generation and later trace generation should be altered accordingly - more on this in Section 6.5.

6.2.4 Trace generator

In Figure 6.5, the idea of the trace generator of the MoVES tool is shown.

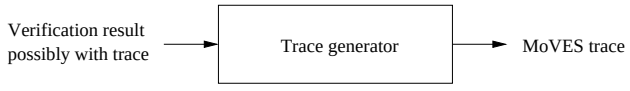


Figure 6.5: The trace generator

Trace generation by the MoVES tool is rather straightforward, but highly depends on the output of the verification backend. This explanation will be conducted in terms of the output produced by the UPPAAL model checker. If a different verification backend is used, the approach will differ - more on this in Section 6.5. The general idea of trace generation can be understood from the explanation in this section.

The output of a UPPAAL verification is a result (i.e. *Property is satisfied* or *Property is NOT satisfied*). Furthermore, an option can be enabled to generate a counter-example trace in addition to the verification result. This trace is given in terms of the UPPAAL timed-automata syntax and state space, and it can be rather tedious to examine this directly. Therefore, the trace generator parses this trace and provides the user with a diagram that is given in terms of the original system specification with time points for task releases, task executions and deadline misses. This diagram resembles that of figure 3.3. We call such a diagram a *MoVES trace*.

The trace provided by the UPPAAL verification contains a snapshot of each state and each transition that takes the combined network of timed automata from the initial state to a state where the verification property (`A[] !missedDeadline`) is violated. In generating a trace in the form of a MoVES trace, the trace generator parses the UPPAAL and represents the trace in suitable data structures.

Functions over these data structures can then identify which tasks were in what locations at any given time point in the trace. This is exactly the information needed in order to generate a MoVES trace. Table 6.1 shows how task states correspond to the locations in the stop-watch automata from Figure 5.11. Also, the corresponding symbol used in a MoVES trace is given for each of these task states.

The MoVES trace corresponding to that from Figure 3.3 is given here:

```
      | 0 2 4 6 8 0 2 4 6 8 0 2 4 6 8 0 2 4 6 8 0 2 4 6
T1    | ++  ++  ++  ++  ++  ++  ++  ++  ++  ++  ++  ++
T2    | OO+   +     OO+   +     OO+   +     OO+   +
T3    | OOOO++OO++  OOOO++OO++  OOOO++OO++  OOOO++OO++
T4    |                                         OO++OOX
T2_T3 | OOO+  O+    OOO+  O+    OOO+  O+    OOO+  O+
```

Note that `T2_T3` corresponds to the activity on the bus b_1 from Figure 3.3, due to the fact that the communication from `T2` to `T3` is the only activity on the bus.

On a side note, the original trace from UPPAAL representing this trace is 684,545 characters on 3,148 lines of text. This emphasizes how much less tedious it is to analyze the representation of the trace that the MoVES trace provides.

A lexer definition, a parser definition and a collection of SML functions conducts the parsing of the trace from UPPAAL, extracts the information needed in order to generate the MoVES trace and generates the MoVES trace to be presented to the user. The concrete SML implementation of the lexer, parser and these functions can be found in Appendix B.3.

6.3 Integrating the Pieces

The model generator (*modelgen*) and the trace generator (*tracegen*) are two individual programs that require suitable arguments for correct analysis. Furthermore, the frontend and the verification backends are individual parts that need to be executed in the correct manner. Therefore, batch scripts ensure correct invocation of the programs with their arguments and also provide the user with default settings if simply the fastest analysis is desired.

The batch script allows usage of the MoVES tool in the following ways:

- Simple default analysis: For this type of analysis, no option is specified, and the default verification structure and verification backend are used as basis for the analysis.
- Analysis with specific verification structure: For this type of analysis, the user specifies which verification structure to use as basis for the analysis. With the choice of verification structure comes the verification backend selection, as each verification structure is associated with a specific backend.

For either analysis type, the MoVES tool conducts the following actions:

1. The system specified in the MoVES language is parsed and represented in suitable data structures in a MoVES syntax tree.
2. The model generator generates a verifiable implementation based on the verification structure and the specified system. The generated model inherits the file name of the system specification with an added file extension,

".xml" in the case of UPPAAL models. The model generator also generates a verification query file if needed. In the case of UPPAAL verification queries, the query file inherits the file name of the system specification with the added file extension ".q".

3. The verification backend conducts the specified verification and provides the verification result (i.e. *Property is satisfied* or *Property is NOT satisfied*). Furthermore, a trace file is generated by the verification backend. The trace file inherits the file name of the system specification with the added file extension ".trace".

4. The trace generator generates a MoVES trace, which corresponds to the trace file generated by the verification backend, and presents it to the user.

Currently, we have developed two batch scripts, one for Windows users and another for Linux users. Both of these scripts can be found in Appendix C.

We can now show that the idea behind the framework is reached and that it should be easy to make changes to verification structures without the need for recompilation of the MoVES tool. Consider the stop-watch verification structure presented in Section 5.1.4. This structure can be used to do analysis with the MoVES tool. If we now consider the alternative stop-watch verification structure presented in Section 5.1.5 to be a refinement of the original structure, we see that this verification structure can be used by the MoVES tool without any need for recompilation or changes to the tool. We simply use a different option for choosing the correct verification structure.

6.4 The MoVES Tool Available Online

Interested users should direct their attention to the website:

`http://www.imm.dtu.dk/moves`

Here the MoVES tool can be downloaded as platform-specific executables and batch scripts. Also, the full source code written in SML can be downloaded. Note that compiler (*mosmlc*), lexer generator (*mosmllex*) and parser generator (*mosmlyac*) for Moscow ML are needed for compilation of the MoVES tool.

At this website there is also documentation and instructions for MoVES, as well as links to academic work concerning the development and theoretic background.

6.5 Summary

In this chapter, we presented the MoVES tool. The basic use of the tool was shown through an example, and we showed how to use some of the more advanced features, e.g. different verification structures. For more advanced users, we presented the structure of the framework that makes up the MoVES tool. This was intended to provide the users with an overview to make it easier to improve the MoVES tool, e.g. by changing verification structures or introducing different verification backends.

The introduction of different verification backends has been slightly touched upon and can be summed up here. In the work presented in Chapter 5, different verifiable implementations using the Uppaal syntax were provided. It has become clear, through the development of the MoVES tool, that using different versions of the Uppaal model checker does not complicate the development of the MoVES tool much, as almost the same syntax is used.

If a user wishes to introduce a different timed-automata model checker, such as Kronos, the frontend could probably still be used without alterations. In the model generator more changes would be needed, as the syntax used for verification structures and verifiable implementations for Kronos differs from that for Uppaal. But the general idea could be kept, as both deal with networks of timed-automata. It is clear that a totally different trace generator would be needed, as the verification result from Kronos most certainly would differ from Uppaal results.

However, if a user would like to introduce a totally different verification backend - a SAT solver for example - then much more severe alterations to the different parts of the MoVES tool would be needed. It would be wise to develop verifiable implementations in the form of SAT-solving problems, as was done for timed-automata implementations in Chapter 5 using the annotations. Much inspiration can be found in the structure of the current MoVES tool, even for such different backends.

Chapter 7

Examples

In this chapter we will provide some examples of system specification and use the MoVES tool to conduct analyses. The chapter serves as an indication of how MoVES can be used and may inspire further development.

With the examples presented in this chapter, we show how the MoVES tool can be used in connection with analyses on a few different levels. In Section 7.1, we show how the MoVES tool is envisioned to be used in connection with design space exploration. The example of an mp3 decoder in Section 7.2 shows that the MoVES tool can be used for systems of a size that resembles interesting systems found in industry.

In Section 7.3, we show how the MoVES tool can be used to analyze systems with multiprocessor anomalies. The example provided in Section 7.4 shows how the MoVES tool does analysis on systems where a deadline is not missed before much later than the maximal offset and a hyper-period. Finally, in Section 7.5, we experiment with systems with very large hyper-periods, and test the limits of the size of systems that the MoVES tool can conduct analysis on.

All the verification of the examples in this chapter has been conducted on a standard PC running Windows Vista, with a 2.4 GHz Intel Core2 Duo processor and 4GB of RAM. The only exceptions are the experiments with the size of the computation tree in Section 7.5.1. The system that these experiments have been

```
Application
 Task: T1
 Period: 4
 Offset: 0

 Task: T2
 Period: 6
 Offset: 0

 Task: T3
 Period: 6
 Offset: 0

 Task: T4
 Period: 6
 Offset: 40

Dependencies
 T2 -> T3 : 2
```

```
Platform
 Proc: P1
 Sch: RM

 Proc: P2
 Sch: EDF

 Bus: b1
 Arb: FIFO
 Speed: 2

Mapping
 T1 : P1
 T2 : P1
 T3 : P2
 T4 : P2
```

```
Creq
 T1 @ P1
  Bcet: 2
  Wcet: 2

 T2 @ P1
  Bcet: 1
  Wcet: 1

 T3 @ P1
  Bcet: 5
  Wcet: 5

 T3 @ P2
  Bcet: 2
  Wcet: 2

 T4 @ P2
  Bcet: 2
  Wcet: 3

Property
 Schedule?
```

Figure 7.1: MoVES specification for windmill control system

conducted on is specified in that section.

7.1 The Windmill Control System

This example is the windmill control system that we have used several times previously to explain areas of MoVES. The first time we presented this example was in Section 1.2.1. We show the MoVES specification for this system one more time in Figure 7.1 to avoid confusion.

Let the specification from Figure 7.1 be saved in a file, say "windmill_control.mvs". With the following command:

```
moves windmill_control.mvs
```

the following MoVES trace is produced:

```
Verifying property 1 at line 1
 -- Property is NOT satisfied.
Showing counter example.
      | 0 2 4 6 8 0 2 4 6 8 0 2 4 6 8 0 2 4 6 8 0 2 4 6
T1    | ++  ++  ++  ++  ++  ++  ++  ++  ++  ++  ++  ++
T2    | OO+   +     OO+   +     OO+   +     OO+   +
T3    | OOOO++OO++  OOOO++OO++  OOOO++OO++  OOOO++OO++
T4    |                                         OO++OOX
T2_T3 | OOO+  O+    OOO+  O+    OOO+  O+    OOO+  O+
```

Upon examination of the MoVES trace, we observe that the missed deadline occurs after 46 milliseconds and that it is the task `T4` that misses the deadline.

7.1.1 Exploring the design space

With design space exploration as described in Section 3.1.8, three different actions are available in order to avoid the missed deadline and construct a schedulable system: 1) remapping of some tasks to different processing elements, 2) rewriting the application to generate different timely properties for some tasks or alter dependencies, and 3) reconfiguring the execution platform by e.g. changing the protocols used by the different processing elements and busses such as scheduling principles and arbiters.

Remapping

In the case of the windmill control system, there is only one option available for remapping. The task `T3` can be mapped to both processing elements, `P1` and `P2`. By altering the specification, mapping `T3` to `P1`, and conducting schedulability

analysis on the altered specification, we get the following from the MoVES tool:

```
Verifying property 1 at line 1
 -- Property is NOT satisfied.
Showing counter example.
   | 0 23
T1 | ++
T2 | OO+
T3 | OOOX
T4 |
```

The schedulability analysis fails after only 3 milliseconds with a missed deadline of `T3`. We conclude that remapping alone is not a suitable action for constructing a schedulable system. (Remapping could possibly be done in connection with one of the other actions in design space exploration.)

Rewriting the application

If it is possible to rewrite the application so that the timely properties of some tasks change, it may generate a schedulable system. If, for example, it is possible to rewrite the task `T4` to have both best-case and worst-case execution time be 2, we can conduct schedulability analysis on the altered system. By altering the worst-case execution time for `T4` from 3 to 2 in the original specification and conducting analysis on the altered system, we get the following from the MoVES tool:

```
Verifying property 1 at line 1
 -- Property is satisfied.
```

The altered system is schedulable. We can conclude that if `T4` can be implemented so that both best-case and worst-case execution times are 2, then the system is without deadline misses. However, we find that it is not possible to implement `T4` to exhibit such behavior. We therefore turn to reconfiguration of the execution platform.

Reconfiguring the execution platform

If we can change the configuration of the execution platform, it may generate a schedulable system. We turn to the scheduling principles used by the processing

elements to see if we can gain something there. The processing element P2 uses rate-monotonic scheduling, and if it can use earliest-deadline-first scheduling, a change here could lead to a solution to the schedulability problem. By altering the original specification to let P2 use EDF scheduling and conducting schedulability analysis in the altered system, we get the following from the MoVES tool:

```
Verifying property 1 at line 1
 -- Property is satisfied.
```

The altered system is schedulable. We can conclude that if P2 can use earliest-deadline-first scheduling instead of rate-monotonic scheduling, we can generate a schedulable system by using this.

More reconfiguration

Many factors could be of interest when constructing a system such as the windmill control system. In reconfiguring the execution platform, we changed the scheduling principle of one processing element from rate-monotonic scheduling to earliest-deadline-first scheduling, so that both processing elements are using earliest-deadline-first. However, considering that the administration involved with using earliest-deadline-first is probably greater than that of rate-monotonic, this could lead the system designer to prefer rate-monotonic scheduling.

We found that using earliest-deadline-first on P2 led to the system being schedulable, but we did not explore the scheduling principle of P1. We now attempt to change the system specification so that the scheduling principle of P1 is EDF instead of RM in the already-altered specification where P2 uses EDF. Conducting schedulability analysis on this system, we get the following from the MoVES tool:

```
Verifying property 1 at line 1
 -- Property is satisfied.
```

We can conclude that by altering the system from the original specification so that P1 uses rate-monotonic scheduling and P2 uses earliest-deadline-first scheduling, we can generate a schedulable system.

task	period π_τ	execution time $bcet_\tau = wcet_\tau$	offset o_τ
τ_0	30,000	45	0
τ_1	30,000	20	0
τ_2	30,000	20	0
τ_3	30,000	1,545	0
τ_4	30,000	1,545	0
τ_5	30,000	595	0
τ_6	30,000	595	0
τ_7	30,000	2,685	0
τ_8	30,000	108	0
τ_9	30,000	108	0
τ_{10}	30,000	895	0
τ_{11}	30,000	895	0
τ_{12}	30,000	6,087	0
τ_{13}	30,000	6,087	0
τ_{14}	30,000	11,200	0
τ_{15}	30,000	11,200	0

Table 7.1: Timely properties for mp3 decoder

7.2 MP3 Decoder

We now turn to the example of an mp3 decoder briefly introduced in Chapter 2. In Figure 2.5 we provided a task graph with an indication of the mapping onto a platform consisting of two processing elements and a bus connecting them. In [48], a table is provided giving all the timely properties of the system. It quickly becomes clear, however, that the system as a whole cannot be verified using the MoVES tool with UPPAAL as backend, as the system runs out of memory.

In order to make schedulability analysis of the mp3 decoder using the MoVES tool, we remove the non-determinism from choices of execution times by only analyzing worst-case execution times. In Table 7.1 the timely properties for the mp3 decoder in this context is given.

In Figure 7.2, the full MoVES specification for the mp3 decoder is given.

Because the verification still runs out of memory, we remove the inherent non-determinism. See more in Section 5.3.1 on non-determinism introduced by modelling using timed automata composed in parallel. With the specification given in Figure 7.2 located in the file called "mp3Dec.mvs", the schedulability analysis

```
Application
 Task: T0
 Period: 30000
 Offset: 0

 Task: T1
 Period: 30000
 Offset: 0

 Task: T2
 Period: 30000
 Offset: 0

 Task: T3
 Period: 30000
 Offset: 0

 Task: T4
 Period: 30000
 Offset: 0

 Task: T5
 Period: 30000
 Offset: 0

 Task: T6
 Period: 30000
 Offset: 0

 Task: T7
 Period: 30000
 Offset: 0

 Task: T8
 Period: 30000
 Offset: 0

 Task: T9
 Period: 30000
 Offset: 0

 Task: T10
 Period: 30000
 Offset: 0

 Task: T11
 Period: 30000
 Offset: 0

 Task: T12
 Period: 30000
 Offset: 0

 Task: T13
 Period: 30000
 Offset: 0

 Task: T14
 Period: 30000
 Offset: 0

 Task: T15
 Period: 30000
 Offset: 0

 Dependencies
  T0 -> T1 : 0
  T0 -> T2 : 0
  T1 -> T3 : 0
  T2 -> T4 : 0
  T3 -> T5 : 0
  T4 -> T6 : 0
  T5 -> T7 : 0
  T6 -> T7 : 0
  T7 -> T8 : 0
  T7 -> T9 : 0
  T8 -> T10 : 0
  T9 -> T11 : 0
  T10 -> T12 : 0
  T11 -> T13 : 0
  T12 -> T14 : 0
  T13 -> T15 : 0

Platform
 Proc: P1
 Sch: RM

 Proc: P2
 Sch: RM

 Bus: B1
 Arb: FIFO
 Speed: 2

Mapping
 T0 : P1
 T1 : P1
 T2 : P2
 T3 : P1
 T4 : P2
 T5 : P1
 T6 : P2
 T7 : P2
 T8 : P2
 T9 : P1
 T10 : P2
 T11 : P1
 T12 : P2
 T13 : P1
 T14 : P2
 T15 : P1

Creq
 T0 @ P1
  Bcet: 45
  Wcet: 45

 T1 @ P1
  Bcet: 20
  Wcet: 20

 T2 @ P2
  Bcet: 20
  Wcet: 20

 T3 @ P1
  Bcet: 1545
  Wcet: 1545

 T4 @ P2
  Bcet: 1545
  Wcet: 1545

 T5 @ P1
  Bcet: 595
  Wcet: 595

 T6 @ P2
  Bcet: 595
  Wcet: 595

 T7 @ P2
  Bcet: 2685
  Wcet: 2685

 T8 @ P2
  Bcet: 108
  Wcet: 108

 T9 @ P1
  Bcet: 108
  Wcet: 108

 T10 @ P2
  Bcet: 895
  Wcet: 895

 T11 @ P1
  Bcet: 895
  Wcet: 895

 T12 @ P2
  Bcet: 6087
  Wcet: 6087

 T13 @ P1
  Bcet: 6087
  Wcet: 6087

 T14 @ P2
  Bcet: 11200
  Wcet: 11200

 T15 @ P1
  Bcet: 11200
  Wcet: 11200

Property
 Schedule?
```

Figure 7.2: MoVES specification for mp3 decoder

using the MoVES tool can be done. We use the genuine discrete model as a basis and remove the inherent non-determinism. The analysis can be conducted with the command:

```
moves -nc-nt mp3Dec.mvs
```

The result of the schedulability analysis from the MoVES tool is:

```
Verifying property 1 at line 1
 -- Property is satisfied.
```

The system as specified is schedulable, i.e. the mp3 decoder will meet all of its deadlines in all of its periods if all tasks are executed in worst-case execution time.

7.2.1 Re-introducing some non-determinism

Although it is not possible to analyze the original mp3 decoder with all the non-determinism included (i.e. all intervals from best-case to worst-case execution times), we can still add a little of the non-determinism. We add one time unit of non-determinism for each task that has worst-case execution time of more than 500 time units (i.e. all tasks except `T0`, `T1`, `T2`, `T8` and `T9`). In Table 7.2 the timely properties of the system with that non-determinism reintroduced are given.

Analysis of the mp3 decoder with some non-determinism can now be conducted using the MoVES tool, and the result is:

```
Verifying property 1 at line 1
 -- Property is satisfied.
```

This example shows that although the MoVES tool cannot analyze the original mp3 decoder with all its non-determinism, it is possible to conduct analysis of the system with some of the non-determinism included.

task	period π_τ	best case $bcet_\tau$	worst case $wcet_\tau$	offset o_τ
τ_0	30,000	45	45	0
τ_1	30,000	20	20	0
τ_2	30,000	20	20	0
τ_3	30,000	1,544	1,545	0
τ_4	30,000	1,544	1,545	0
τ_5	30,000	594	595	0
τ_6	30,000	594	595	0
τ_7	30,000	2,684	2,685	0
τ_8	30,000	108	108	0
τ_9	30,000	108	108	0
τ_{10}	30,000	894	895	0
τ_{11}	30,000	894	895	0
τ_{12}	30,000	6,086	6,087	0
τ_{13}	30,000	6,086	6,087	0
τ_{14}	30,000	11,199	11,200	0
τ_{15}	30,000	11,199	11,200	0

Table 7.2: Timely properties for mp3 decoder

7.3 Multiprocessor Anomalies

In this section we examine the example system with an interesting multiprocessor anomaly that was shown in Section 3.1.11. The MoVES specification of this system is given in Figure 7.3 once again to avoid confusion.

Conducting schedulability analysis with the MoVES tool on the system yields the following result:

```
Verifying property 1 at line 1
 -- Property is NOT satisfied.
Showing counter example.
   | 0 23
T1 | +  +
T2 | O+ O
T3 | +  +
T4 | OO+O
T5 | OOOX
```

Note that the counter-example trace is consistent with the expected trace shown in Figure 3.5(b), namely that `T5` misses a deadline after three time units.

```
Application
 Task: T1
 Period: 3
 Offset: 0

 Task: T2
 Period: 3
 Offset: 0

 Task: T3
 Period: 3
 Offset: 0

 Task: T4
 Period: 3
 Offset: 0

 Task: T5
 Period: 3
 Offset: 0

Dependencies
 T1 -> T2 : 0
 T3 -> T4 : 0
 T4 -> T5 : 0
```

```
Platform
 Proc: P1
 Sch: RM

 Proc: P2
 Sch: RM

 Proc: P3
 Sch: RM

 Bus: b1
 Arb: FIFO
 Speed: 2

Mapping
 T1 : P1
 T2 : P2
 T3 : P3
 T4 : P2
 T5 : P3
```

```
Creq
 T1 @ P1
  Bcet: 1
  Wcet: 2

 T2 @ P2
  Bcet: 1
  Wcet: 1

 T3 @ P3
  Bcet: 1
  Wcet: 1

 T4 @ P2
  Bcet: 1
  Wcet: 1

 T5 @ P3
  Bcet: 1
  Wcet: 1

Property
 Schedule?
```

Figure 7.3: MoVES specification for system with multiprocessor anomaly

If the application is rewritten so that T1 always has an execution time of two time units, schedulability analysis of the altered system can be conducted. Conducting this analysis using the MoVES tool gives the following result:

```
Verifying property 1 at line 1
 -- Property is satisfied.
```

This example illustrates the interesting phenomenon of a local best-case execution time triggering a missed deadline, where worst-case execution time does not (i.e. this is an example of a multiprocessor anomaly).

7.4 Very Late Deadline Miss

We now turn our attention to an example that shows just how late a deadline miss can occur. The system was introduced in Section 4.7.2. In Figure 7.4, the MoVES specification for this system is given.

```
Application
 Task: T1
  Period: 3
  Offset: 0

 Task: T2
  Period: 3
  Offset: 1

 Task: T3
  Period: 3
  Offset: 2

 Dependencies
```

```
Platform
 Proc: P1
  Sch: EDF

 Bus: b1
  Arb: FIFO
  Speed: 2

Mapping
 T1 : P1
 T2 : P1
 T3 : P1
```

```
Creq
 T1 @ P1
  Bcet: 1
  Wcet: 1

 T2 @ P1
  Bcet: 1
  Wcet: 1

 T3 @ P1
  Bcet: 2
  Wcet: 2

Property
 Schedule?
```

Figure 7.4: MoVES specification of system with late deadline miss

Conducting schedulability analysis using the MoVES tool provides the following

result:

```
Verifying property 1 at line 1
 -- Property is NOT satisfied.
Showing counter example.
   | 0 2 4 6 8 01
T1 | +  O+ OO+OOO
T2 |  +  O+ OO+OO
T3 |   ++ O++OO+X
```

We can see that the system is not schedulable as `T3` misses a deadline after 11 time units. The interesting aspect here is that the maximal offset is two and the hyper-period is three. In other words, the deadline miss occurs in the third hyper-period after the maximal offset.

7.5 Systems with Large Hyper-Periods

In this example, we examine a system with a very large hyper-period. We introduced the example in Section 4.7.4. The timely properties for this system were given in Figure 4.4. In Figure 7.5, we provide the MoVES specification for the system.

```
Application
 Task: T1
  Period: 11
  Offset: 0

 Task: T2
  Period: 8
  Offset: 10

 Task: T3
  Period: 251
  Offset: 27

 Dependencies
```

```
Platform
 Proc: P1
  Sch: EDF

 Bus: b1
  Arb: FIFO
  Speed: 2

Mapping
 T1 : P1
 T2 : P1
 T3 : P1
```

```
Creq
 T1 @ P1
  Bcet: 1
  Wcet: 3

 T2 @ P1
  Bcet: 1
  Wcet: 4

 T3 @ P1
  Bcet: 1
  Wcet: 8

Property
 Schedule?
```

Figure 7.5: MoVES specification of system with large hyper-period

With periods 11, 8 and 251, the hyper-period is calculated as LCM$\{11, 8, 251\} =$ 22088.

Conducting schedulability analysis on this system using the MoVES tool tests the capabilities of the MoVES framework and the verification backend. The result is the following:

```
Verifying property 1 at line 1
 -- Property is satisfied.
```

However, if we attempt to analyze the system where `T3` has a worst-case execution time of nine instead of eight, the verification fails with an *out of memory* error message. This system tests the limits of how large a system the MoVES framework can analyze when using UPPAAL as verification backend.

Note that the verification was conducted on a standard PC running Windows Vista. Since the UPPAAL model checker (*verifyta*) is a single-threaded 32-bit process, it can only be allocated 2GB of RAM by a Windows operating system. If running the MoVES tool with the UPPAAL backend on a PC running Linux, the 32-bit process can be allocated in approximately 3.4GB of RAM, and somewhat larger systems can be verified that way. We now look a bit further into this.

7.5.1 Experimenting with size of computation tree

We will now conduct some experiments to analyze the correlation between the size of the computation tree of a system needing to be explored (see Section 4.7.3 for details concerning sizes of computation trees), as compared to the amount of memory used in schedulability analysis with the MoVES tool.

The following verification examples have all been conducted on Linux servers with 4 dual core AMD Opteron processors running at 2.4 GHz and with 32GB of RAM.

The example with a large hyper-period is now analyzed in eight slightly different versions, where we change the worst-case execution time for the task `T3`. In Table 7.3 we show the number of nodes in the computation tree at the depth it suffices to search when checking for schedulability as shown in Section 4.7.3 with the definition of *maxChoices* in Equation (4.10). Also, we show the maximal amount of memory used by the MoVES tool when conducting schedulability analysis.

$wcet_{T3}$	$maxChoices$	Max memory used
5	$1.5 \cdot 10^{13}$	1.3GB
8	$2.4 \cdot 10^{13}$	1.8GB
11	$3.3 \cdot 10^{13}$	2.2GB
14	$4.2 \cdot 10^{13}$	2.4GB
17	$5.1 \cdot 10^{13}$	2.6GB
20	$6.0 \cdot 10^{13}$	2.9GB
23	$6.9 \cdot 10^{13}$	3.2GB
26	$7.8 \cdot 10^{13}$	3.4GB

Table 7.3: Size of computation tree vs. memory usage by the MoVES tool

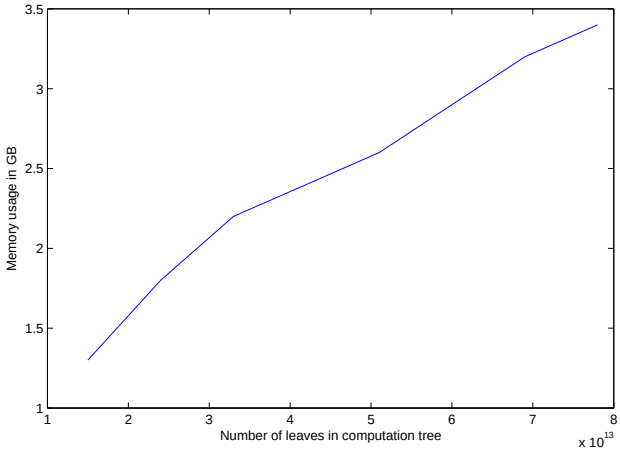


Figure 7.6: Computation tree vs. memory usage

We see that as the size of the computation tree grows, the amount of memory needed by the MoVES tool when conducting schedulability analysis grows somewhat accordingly. This is an indication that there is a correlation between the memory requirements needed for analysis using the MoVES tool and the computation tree identified in the formal model in Chapter 4.

7.6 Summary

In this chapter we have shown how systems can be analyzed and schedulability verified using the MoVES tool. The examples showed that the framework can

conduct analysis of systems that are close to industrially-interesting sizes such as the mp3 decoder. We also showed how the MoVES tool can be used to analyze systems with multiprocessor anomalies. In short, this chapter gave a snapshot of how MoVES can be used in the design space exploration at an early stage in the design process of embedded systems.

Chapter 8

Perspective

In this chapter we will highlight some of the perspective enabled through the work presented in the dissertation. Firstly in Section 8.1, we discuss the use of different verification structures and backends in the MoVES tool. In Section 8.2, we comment on analysis of purely deterministic systems where all tasks have $bcet = wcet$. Section 8.3 contains ideas regarding analysis of resource usage such as power and memory consumption. Then, in Section 8.4, we have a deeper explanation of the realization of some tasks of embedded systems in hardware and how the realized components can have timely properties from the MoVES specification verified.

In Section 8.5, we touch on some of the underlying assumptions that are integrated in the model for MoVES, and we give indications of how some of these can be lifted and the model altered accordingly. Section 8.6 discusses how the task model in MoVES relates to tasks found in networked, embedded control systems. We especially touch on control tasks, where sampling is an integrated component. Finally, in Section 8.7, we present a vision for a development process for embedded systems that relies on refinement steps. We explain where we feel that MoVES fits into this type of a process.

8.1 Verification Structures and Backends

The work presented in this dissertation relies on the use of four specific verification structures. They are all timed automata in UPPAAL syntax. Two verification backends are used, which are two different versions of the UPPAAL model checker.

The MoVES tool is developed as a framework to be independent of verification structures and backends. In Chapter 6 we discussed how the use of other verification structures and backends can be deployed within the framework, and the ease with which verification structures can be altered and still be used within the framework without the need for recompilation.

Interesting alternative backends could include other timed-automata model checkers such as Kronos [20], different SAT solvers such as iSAT [37] and implementation of well-known tree search algorithms. The possible use of Kronos as backend and the implications thereof were discussed in Section 6.5.

If using a SAT solver as verification backend for the MoVES tool is desired, suitable verification structures should be developed. Posing the schedulability problem as a SAT problem can be conducted, as an upper limit of the depth of the computation tree has been defined. Therefore, bounded model checking could be used to conduct automatic verification of such a problem in the MoVES tool.

Another approach for solving the schedulability problem would be to generate the computation tree directly. On the basis of this, well-known tree search algorithms (e.g. depth-first or breadth-first search) could be used to conduct the analysis. This approach could contribute as another verification backend to the MoVES tool.

8.2 Purely Deterministic Systems

The MoVES language, the formal model and the MoVES tool presented in this dissertation have focused on capturing systems that exhibit behavior, which can be represented in a computation tree such as the one depicted in Figure 4.2. This type of computation tree is really interesting if some branching occurs. However, from the point of analysis, even systems that exhibit a behavior that can be represented in a non-branching computation tree may still give valuable verification results. We call such systems purely deterministic, as the computa-

tion tree never branches, i.e. the system is without non-deterministic choices of execution times.

In the case of the mp3 decoder that we conducted analysis on in Section 7.2, we did the initial analysis on a system where the computational requirements for all tasks had best-case execution time equal to worst-case execution time. Systems like this exhibit exactly the behavior that can be represented in a non-branching computation tree. Verification results for schedulability analysis of such systems are still correct and valuable.

An interesting concept occurs when conducting verification on a non-branching computation tree. There is no need to keep track of which nodes in the computation tree have been visited already, and therefore, the memory usage of such a verification can be dramatically reduced. As a result, much larger systems can be verified if they are represented in a non-branching computation tree.

In [46], we did experiments with this type of model. An experimental version of UPPAAL, where no tracking of visited nodes in the computation tree needed to be kept, was made available to us. With the use of this as verification backend, we conducted schedulability analysis on a smartphone consisting of 3 applications with a total of 103 tasks executing on a platform with 4 cores. This can easily be considered a system that is close to industrially-interesting size.

8.3 Analysis of Resource Usage

In MoVES, as presented in this dissertation, no specific attention has been given to the area of resource analysis such as analysis of power consumption and memory usage. However, we have conducted several experiments where resource analysis was a component.

The general idea of introducing resource usage in MoVES is to add the attributes identifying the individual resource usage by specific tasks on specific processing elements directly to the computational requirements. Furthermore, the notion of allocation of resources must be introduced, so that access to resources is managed. The introduction of allocation of resources to MoVES is envisioned to be conducted much like the scheduling. Different protocols for managing resource allocation (e.g. priority ceiling) can be modelled.

With the power consumption of executing a task on a given processing element specified, suitable verification queries can give upper and lower bounds on the total power consumption of the system over time. The same can be done for

memory usage, and analysis can identify the greatest amount of memory needed at any time for any possible execution of the system.

In [46], we provided examples of analysis including resource usage. We showed that bounds on power consumption and memory usage could be guaranteed using an extended model in the general MoVES framework.

8.4 Hardware Specifications and Tasks in MoVES

The notion of a task in MoVES is defined in terms of its timely properties and dependencies with other tasks. When mapped to a specific processing element, the computational requirements for the task with that mapping must be specified. We have not given any explanation of how to extract the computational requirements (i.e. best-case and worst-case execution times) as of yet.

In this dissertation, we have not touched the topic of whether tasks are implemented in hardware or software. This has been done so that developers are not forced to make design decisions too early and to leave room for design space exploration. If a task is implemented in software, one can attempt executing that specific task on different processing elements in order to extract computational requirements; this is how the extraction of computational requirements for the mp3 decoder example from Section 7.2 was done in [60].

With tasks implemented strictly in hardware, there can be more direct ways to extract computational requirements. This is the case if these tasks are implemented as dedicated hardware components (ASIC) or as programmable components (FPGA). Extraction of computational requirements on the basis of specifications of tasks implemented in hardware requires a language with a certain abstraction level. The level should be low enough to be realized, i.e. automatically synthesized in hardware, but also high enough to avoid a level of detail that will clutter a clear semantics. We have identified Gezel as being such a language.

8.4.1 Gezel specifications

Gezel [59] is a high-level hardware description language. It comes with an interpreter as well as a translator with VHDL [36] as its target language. With the interpreter, simulation can be done at the level of abstraction provided by Gezel, a level that is higher than other hardware description languages (e.g.

VHDL and Verilog [62]). This is very useful for debugging. The translator can give synthesizable VHDL descriptions. In other words, a Gezel description is a hardware specification that can be realized.

A Gezel specification describes a number of *components* and their interconnections. A component in Gezel is made up of a *datapath* and a *controller*. A datapath provides a number of *actions*, and a controller is expressed as a *finite state machine* where one or more actions can be executed in each state transition.

This model is called a *finite state machine with datapath* (FSMD). Figure 8.1 shows the structure of an (FSMD).

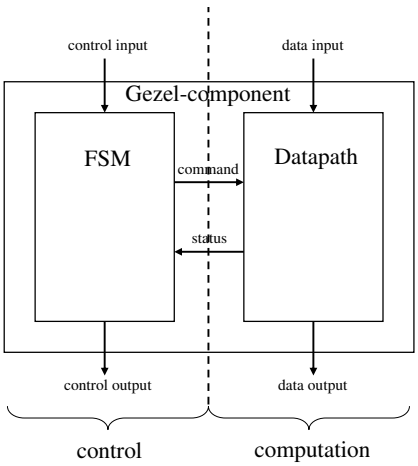


Figure 8.1: The FSMD model

8.4.2 Key elements from the semantics domain

The Gezel language as described in [59] does not have formal semantics, and there are no tools for verification of Gezel specifications. In [27], we gave a semantics domain that can be used for hardware design languages like Gezel. Here, we will highlight some of the key notions of the semantics domain but we refer to [27] for details.

The semantics provide definitions for *modules*. Modules are the building blocks for systems. There are three different types of modules: *basic modules*, *composition of modules* and *top-level module*.

Semantically, a module is basically a Mealy automaton with the addition of a single function that allow for parallel composition of modules, especially for cases where inputs are computed by other modules. Composition of modules preserves this semantical basis.

A basic module consists of input- and output ports, signals, registers, and a set of actions. Each action is a single assignment program where registers, signals and output ports can be assigned a value. All assignments of registers and signals must be done in basic modules. No such assignments can exist at the level of composition of modules or at the top level.

Composition of modules (either basic- or already composed modules) is defined as modules composed in parallel using single assignment programs to connect their input- and output ports as well as any input- or output ports of the resulting composed module.

A top-level module describes the system as a whole at the highest level in the hierarchical structure. All input- and output ports of the modules composed by the top-level module are connected to either ports of the other modules involved in the composition or to the ports of the top-level module. The ports of the top-level module describe the interface (input- and output ports) of the described system to the outside world.

8.4.3 Verification and analysis of quantitative properties

Using the semantics domain highlighted here, an implementation can be constructed on this basis. In [27], we gave such an implementation in finite automata. With the use of the Uppaal model checker, we conducted verification of the functionality of a few smaller examples. The verification could guarantee that any combination of input to the system would generate an output (i.e. a "final" state would be reached), and furthermore that the resulting output was correct (i.e. corresponded to a mathematical specification or result table).

In [11], the type of analysis of implementations in finite automata based on the semantics domain was extended to also include non-functional aspects. We conducted analysis on the size of the resulting hardware component in terms of the required number of lookup-tables on an FPGA. Analysis of the latency and throughput of the resulting hardware component was also conducted.

We showed that the analysis can guarantee computational requirements in terms of number of cycles needed before a result is ready, as well as the latency of one

cycle. This indicates that a link to best-case and worst-case execution times of tasks described as hardware components in Gezel has been established.

8.5 Ideal Assumptions

In the work described in this dissertation, some underlying ideal assumptions have been made. An area for further development could be to examine whether any of these could be lifted and then handled within the framework.

One obvious assumption is the progress of time. On the multi-core systems modelled in MoVES, all clocks in the different components of the system are assumed to be synchronized. There has been much work analyzing clock drift, and it could be an interesting area in the future to examine how drifting clocks affect the models in MoVES. The work in [3] provides timed-automata models for systems with perfectly periodic digital clocks as well as different types of clocks with non-perfect periods. Inspiration for including clocks that drift could possibly be found here.

Another assumption is that the time conducting scheduling changes and task preemption (and possible context switching) is ignored. This time could be assumed to already be included in the computational requirements for the tasks. But since tasks can have several preemptions during one period, and different amounts of preemptions in different periods, it could be very interesting to see how it would affect the models in MoVES if scheduling delay and delay for task preemption and context switches would be included. This is another interesting area for future work.

8.6 MoVES in the Context of Networked Embedded Control Systems

In [15], Cervin el al. present the tool TrueTime, which is a simulation tool that can be used in development of networked embedded control systems. Its main focus is systems developed for automatic control. Often, sampling is an integrated component of the control systems. Specifically, sampled control theory is used, where a discrete digital controller controls a continuous system through sampling.

In sampling control theory, the sampling tasks are usually periodic with a certain

frequency, which is chosen high enough to ensure the correct behavior of the controlled system. Also, there is a natural list of tasks to be done periodically.

It seems that the notions of periodicity and task dependencies are identified in much the same manner in TrueTime and sampled control theory as envisioned in MoVES. The connection to sampled control theory was identified quite late in the project. Most of the time was spent relating the work to streaming applications. As a result, we have not explored this connection further as of yet. It would be an interesting area for further research to explore whether systems from sampled control theory would be naturally modelled and analyzed using MoVES.

8.7 MoVES in a Greater Development Process

Throughout this dissertation, we have pointed out that it is the intent for MoVES to be used by designers of embedded systems in early development phases. We have also shown how to specify systems in MoVES, and through this, described what is needed in order to use the MoVES tool for analysis. We will now discuss how we envision MoVES being used in the development process of embedded systems. The approach envisioned here relies on the notions of specification and refinement from Unifying Theories of Programming [34].

8.7.1 An example - railway crossing refinement

The aim would be to have a development process with a special focus on traceability from an overall requirements specification to components on a multi-core platform. Think of a railway crossing: one would have clear requirements specified mathematically - requirements that specify the timing properties at the highest level of abstraction. These requirements would dictate how signal lights and train gates should act in any successful realization.

The realization of such requirements would be achieved through a number of refinement steps. Each step would contain computational rules that ensure the step preserves the specification from which the refinement was based.

The traceability through the development process would allow developers to clearly identify which parts of the realized components originate from what parts of the specifications at any given refinement step. For example, for a task for opening a train gate at a train crossing, which is executed on a specific

processing element, the developer would be able to identify which part of the initial overall requirements specification gave rise to the task, and exactly what refinement steps led to the exact realization.

8.7.2 Development process in general terms

Initially, an embedded system should ultimately be specified at a high level of abstraction in a simple and convincing fashion using mathematical notation. Through a number of refinement steps, more and more design decisions should be made, and each step should preserve the properties of the specification that are refined.

At some step, one or more applications of the embedded system are specified, including a number of tasks, and task graphs indicating dependencies. At this point, design decisions regarding choices of execution platform can be made. With these choices, MoVES can be used to analyze the different designs, and the MoVES tool can verify that the design decisions preserve the properties of the original specification.

After this refinement step, the result is specifications of the tasks and processing elements that constitute a verifiably correct system based on the analysis conducted with the MoVES tool. Further refinement steps should lead to implementations of the specified tasks, either in software or as dedicated hardware components. The tasks implemented in software are to be executed on processing elements with the chosen real-time operating system attributes. The tasks implemented as dedicated hardware components could possibly be refined toward Gezel-like specifications as mentioned in Section 8.4.

Chapter 9

Conclusion

The general goal of this work is to provide languages, models, tools and methodologies, which in early stages of the design process can help the designer of embedded systems analyze different configurations and setups of systems.

In Chapter 1, we gave an overview of the setting in which this work should be considered, and we pointed out different approaches and current research within the area of analysis of embedded systems. Chapter 2 highlighted the ARTS framework, which has been a key inspiration in the work on MoVES. Through examples, we also showed that for some systems, theories that are well founded within classic scheduling theory for single-processor systems do not generalize to multi-core systems, e.g. the use of a critical instant.

The MoVES language was defined in Chapter 3, together with a few examples that highlighted its use. In Chapter 4, a formal model for MoVES was derived. A computation tree for a system was defined to capture all possible runs of the system. A definition of the schedulability problem in the context of the formal model was given through a decidability result. This gave an upper limit to how much of the computation tree for a system needed to be examined in order to ensure schedulability. A small example system indicated that even apparently simple systems can be difficult to analyze complexity wise.

Chapter 5 showed how the formal model and the schedulability problem could

be encoded using timed automata and verified using the UPPAAL model checker. In using UPPAAL as verification backend, we found that it scaled much better than expected. We learned a few tricks, especially to bound all integer variables and use constants whenever possible to reduce the state space. We also experienced that priorities on processes, which we touched on in Section 5.3.1, was a convenient approach to removing inherent non-determinism. Furthermore, when developing the genuine discrete verification structure from Section 5.1.7, we found that the UPPAAL model checker also performed well for models without clocks. Actually in most cases, analysis based on this verification structure uses less memory for the verification and provides the verification results faster than any of the other verification structures mentioned.

The MoVES tool, a framework for analysis and verification of embedded systems, was presented in Chapter 6. The tool was built to support automatic verification of properties of embedded systems such as timing properties, i.e. schedulability analysis. The tool uses the MoVES language as specification language, and the verification is conducted on implementations of the formal model for MoVES.

In Chapter 7, we used a few examples to show how the MoVES tool works in connection with analysis of embedded systems. We used the MoVES tool in connection with simple design space exploration, but we also showed how it can be used to analyze systems with interesting properties such as multiprocessor anomalies. Finally, some experiments showed the limits on the size of problems that the MoVES tool can analyze.

Chapter 8 gave some insight into how MoVES can be further developed, areas of particular interest when further development is done, and also how we envision MoVES being used in a greater development process when designing embedded systems in the future. Regarding special interest in future development, we should mention the following: 1) analyze and experiment with systems from the area of networked, embedded control systems to examine whether these could directly be analyzed using MoVES and 2) explore how this work could inspire a development process for embedded systems that is closer to classic development methods used in software development - possibly inspired by Unifying Theories of Programming.

9.1 Final Remarks

We have established a semantic basis for analysis of applications executing on multi-core platforms. On this semantic basis, a language, a model and a tool

have been developed for automatic verification in the context of schedulability analysis of multi-core embedded systems. They have also opened up for other types of analysis of such systems. Examples have shown that timed-automata implementations are suitable as verification structures that can form a basis for automatic verification.

With this dissertation we have found and justified, that semantically-based verification is a suitable approach when analyzing embedded systems. A few examples, especially the example of the mp3 decoder in Section 7.2, show great promise in terms of using MoVES to analyze systems that are industrially interesting in size and complexity.

Throughout the dissertation, especially in Chapters 5 and 6, we showed that interesting properties of embedded systems can be modelled and verified using models of timed automata and the UPPAAL model checker.

Appendix A

Timed-Automata Templates for Verification Structures

This appendix includes timed-automata templates for the verification structures presented in the dissertation in Chapter 5.

A.1 Stop-watch automata model

This is the full definition of the stop-watch automata verification structure. The task model was presented in Section 5.1.4.

A.1.1 Global declarations

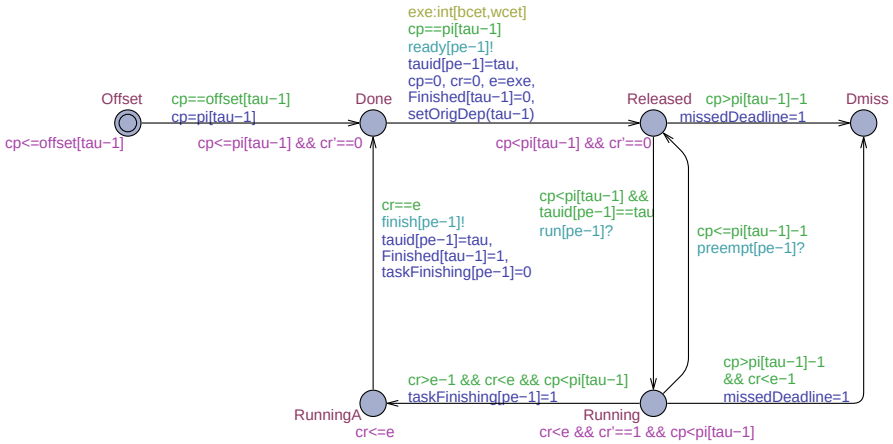
```
1  const int FP=0, RM=1, EDF=2;
2
3  //System-Dependent Decl
4  const int M = ; //The number of Processors
5  const int N = ; //The maximum number of tasks per Processor
6  const int MN = ; //The total number of tasks
7
8  int[FP,EDF] processorScheduling[M] = {}; //Contains information
       about the scheduling principle for each processor.
```

```
9  int[0,MN] onPE[M][N]={}; //global taskids from locals
10 const int pi[MN] = {}; //RM scheduling priorities
11 const int offset[MN] = {}; //global offset information
12 const int fpris[MN] = {}; //FP scheduling priorities
13
14 //array for original dependencies, 1 for dependency , 0 for no
       dependency - {{0,1},{0,0}} means that 1 is dependant on 2
15 bool origdep[MN][MN]={};
16
17 //dynamically updated array for current dependencies
18 bool depend[MN][MN]={};
19
20 int[1,MN] pri[MN]; //EDF scheduling priorities
21
22 //dynamically updated priorities
23 const int NRSteps=, NROffSteps=, MAXOffStep=, MAXStep=;
24
25 const int[0,MAXOffStep] OffSteps[NROffSteps] = {};
26 const int[1,MN] OffPrios[NROffSteps][MN] = {};
27
28 const int[0,MAXStep] Steps[NRSteps] = {};
29 const int[1,MN] Prios[NRSteps][MN] = {};
30
31 const int MaxExe=;
32 const int MaxPi=;
33
34 //System-Independent Decl
35
36 //Synchronization channels
37 broadcast chan reschedule; //broadcast channel for rescheldulingg
       after a task has finished
38 chan synchronize[M], schedule[M];
39 chan ready[M], run[M], preempt[M];
40 chan finish[M];
41
42 int[0,MN] tauid[M]; //transfer of local taskid from task to
       controller
43 int[0,MN] curtid[M]; //variable used to hold the id of the task
       currently chosen
44 //int[0,MN] ftid[M]; //taskid of task which has finished, ftid=0
       means no finished task
45 int[0,MN] ltid[M]; //variable used to hold the id of the task
       currently running
46 bool Released[MN]; //array of tasks which have issued ready signals
47 bool Enabled[MN]; //array of tasks which are not awaiting
       dependencies to be resolved
48 bool rescheduleNeeded; //indicator for the need for a global
       reschedule
49 bool Finished[MN]; //array specifying which tasks are finished now
50 bool WaitDep[MN]; //array for tasks which are awaiting dependencies
       to be resolved
51 bool Running[M]; //indicating wether a task is currently running on
       the processor
52 bool missedDeadline = false; //indicator for a missed deadline
53
```

```
54 //Criteria usable in scheduling
55 int staticCriteria[M][N]; //criterion used for static scheduling
       and contains the original parameters for dynamic scheduling.
56 int dynamicCriteria[M][N]; //criterion used for dynamic scheduling
57
58
59
60 //Locking mechanisms
61 bool disc[MN]; //locking mechanism for ensuring full discretization
       run
62 bool taskFinishing[M]; //locking mechanism for ensuring reaction to
       all finish signals before any ready signals
63
64 //function checking for dependencies for task t
65 bool taskHasDependency(int t) {
66   for (ini : int[0,MN-1]) {
67     if (depend[t][ini]) {
68       return true;
69     }
70   }
71   return false;
72 }
73
74 //function updating dependencies when task t has finished
75 void opdDep(int t) {
76   for (ini : int[0,MN-1]) {
77     depend[ini][t]=false;
78   }
79 }
80
81
82 //function checking for existance of boolean value in array of size
       M
83 bool lockPE(bool la[M]) {
84   for (ini : int[0,M-1]) {
85     if (la[ini]) {
86       return true;
87     }
88   }
89   return false;
90 }
91
92 //function checking for existance of boolean value in array of size
       MN
93 bool lockT(bool pen[MN]) {
94   bool b = false;
95   for (ini : int[0,MN-1]) {
96     if (pen[ini] == true) {
97       return true;
98     }
99   }
100  return false;
101 }
102
103 //function checking if task t is on processing element p
```

```
104 bool isOnPE(int t, int p){
105   for(i:int[0,N-1]){
106     if(onPE[p-1][i]==t) return true;
107   }
108   return false;
109 }
```

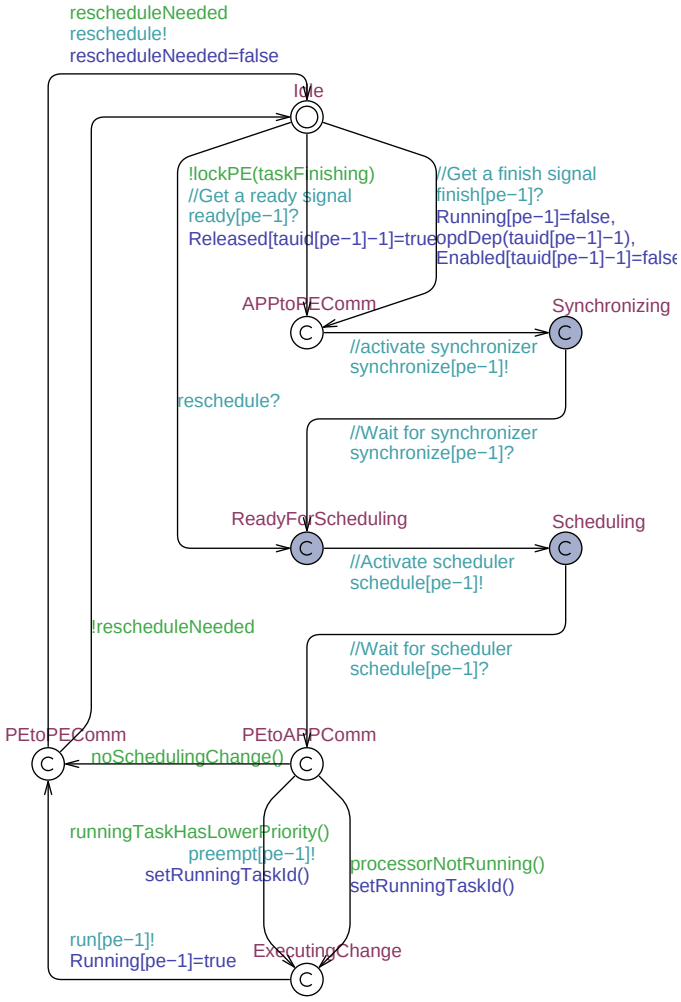
A.1.2 Task template



A.1.3 Task template declarations

```
1  clock cp, cr;
2  int e;
3
4  void setOrigDep(int t) {
5    for (ini : int[0,MN-1]) {
6      if ((offset[t]>offset[ini])&&Finished[ini])
7        depend[t][ini]=false;
8      else
9        depend[t][ini]=origdep[t][ini];
10   }
11 }
```

A.1.4 Controller template

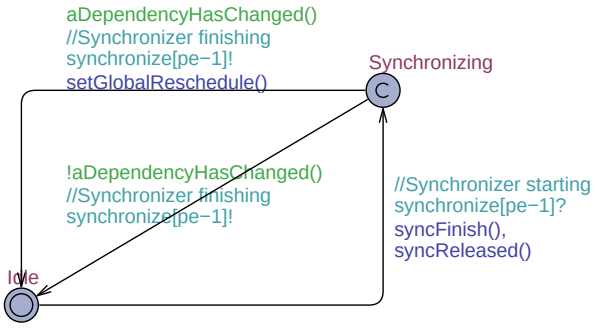


A.1.5 Controller template declarations

```
1  bool processorNotRunning() {
2    return (!Running[pe-1] && curtid[pe-1] != 0);
3  }
4
5  void setRunningTaskId() {
6    tauid[pe-1] = curtid[pe-1];
7    ltid[pe-1] = curtid[pe-1];
```

```
8  }
9
10 bool noSchedulingChange() {
11   return ((ltid[pe-1]==curtid[pe-1] && Running[pe-1]) || curtid[pe
       -1]==0);
12 }
13
14 bool runningTaskHasLowerPriority() {
15   return (ltid[pe-1]!=curtid[pe-1] && Running[pe-1]);
16 }
```

A.1.6 Synchronizer template

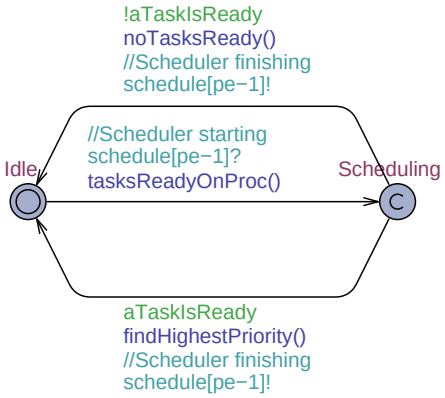


A.1.7 Synchronizer template declarations

```
1  bool depCh; //flag used if a dependency has been changed
2
3
4  bool aDependencyHasChanged() {
5    return (depCh);
6  }
7
8  void setGlobalReschedule() {
9    depCh=false;
10   rescheduleNeeded = true;
11 }
12
13
14
15 void syncFinish() {
16   for (i : int[0,MN-1]) {
17     if (WaitDep[i] && !taskHasDependency(i)) {
18       Enabled[i]=true;
19       WaitDep[i]=false;
20       depCh=true;
21     }
```

```
22    }
23  }
24
25  void syncReleased() {
26    for (i : int[0, MN-1]) {
27      if (Released[i] && isOnPE(i+1,pe)) {
28        Released[i]=false;
29        if (taskHasDependency(i)) {
30          WaitDep[i]=true;
31        }
32        else {
33          Enabled[i]=true;
34        }
35      }
36    }
37  }
```

A.1.8 Scheduler template

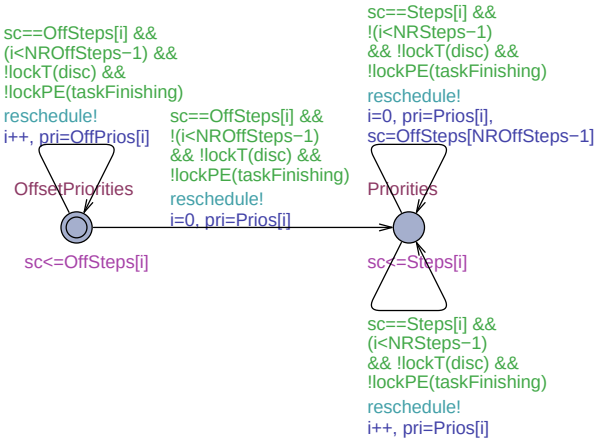


A.1.9 Scheduler template declarations

```
1  int lcri, lcri2; //variables used to hold the criterion of the task
       currently chosen
2  bool aTaskIsReady;
3
4  void noTasksReady() {
5    curtid[pe-1]=0;
6  }
7
8  void findHighestPriority() {
9    for (i : int[0,MN-1]) {
10     if (isOnPE(i+1,pe)) {
11       if (Enabled[i]) {
```

```
12          if (processorScheduling[pe-1] == FP && (fpris[i] < lcri)) {
13            curtid[pe-1]=i+1;
14            lcri=fpris[i];
15          }
16          if (processorScheduling[pe-1] == RM && (pi[i] < lcri || (pi
                [i] == lcri && fpris[i] < lcri2))) {
17            curtid[pe-1]=i+1;
18            lcri=pi[i];
19            lcri2=fpris[i];
20          }
21          if (processorScheduling[pe-1] == EDF && (pri[i] < lcri || (
                pri[i] == lcri && fpris[i] < lcri2))) {
22            curtid[pe-1]=i+1;
23            lcri=pri[i];
24            lcri2=fpris[i];
25          }
26        }
27      }
28    }
29  }
30
31  void tasksReadyOnProc() {
32    aTaskIsReady = false;
33    for (i : int[0,MN-1]) {
34      if (isOnPE(i+1,pe)) {
35        if (Enabled[i]) {
36          aTaskIsReady = true;
37            if (processorScheduling[pe-1] == FP) {
38              lcri = fpris[i];
39            }
40            if (processorScheduling[pe-1] == RM) {
41              lcri = pi[i];
42              lcri2 = fpris[i];
43            }
44            if (processorScheduling[pe-1] == EDF) {
45              lcri = pri[i];
46              lcri2 = fpris[i];
47            }
48            curtid[pe-1]=i+1;
49            return;
50        }
51      }
52    }
53  }
```

A.1.10 Administrating template



A.1.11 Administrating template declarations

```
1  clock sc; //clock for updating dynamically updated priorities
2  int[0,NRSteps-1] i=0;
```

A.1.12 System instantiation

```
1
2  //System-Dependent Inst
3  //Control(pe)
4  //Synchronizer(pe)
5  //Scheduler(pe)
6
7  //Task(pe,tau,bcet,wcet)
8
9  //system Tasks, Cons, Syns, Schs, Tadm;
10
11 //System-Independent Inst
```

A.2 Alternative stop-watch automata model

This is the full definition of the alternative stop-watch automata verification structure. The task model was presented in Section 5.1.5.

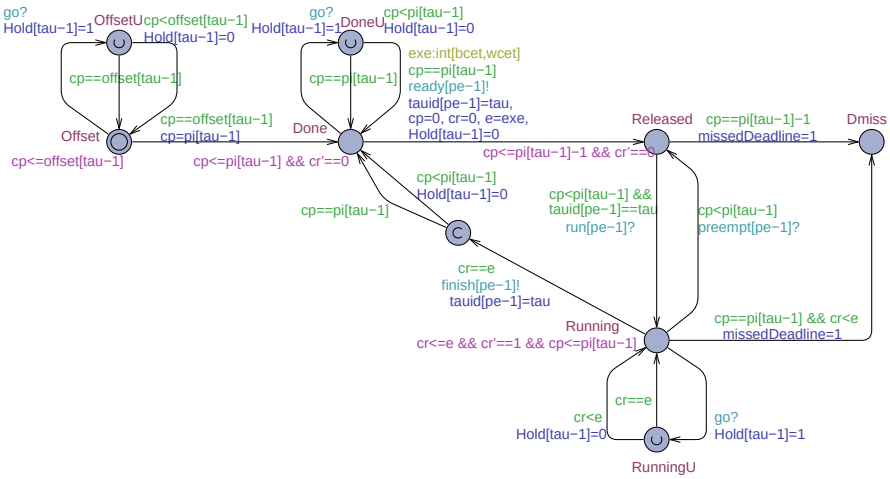
A.2.1 Global declarations

```
1  clock TM;
2  const int FP = 0, RM = 1, EDF = 2; //symbolic representation of
       scheduling principles
3
4  //System-Dependent Decl
5  const int M=; //The number of Processors
6  const int N=; //The maximum number of tasks per Processor
7  const int MN=; //The total number of tasks
8
9  //Contains information about the scheduling principle for each
       processor.
10 const int[FP,EDF] processorScheduling[M] = {};
11 const int[0,MN] onPE[M][N]={}; //tasks on which processors
12
13 const int fpris[MN] = {}; //FP scheduling priorities
14 const int pi[MN] = {}; //RM scheduling priorities
15 const int offset[MN] = {}; //global offset information
16 int[1,MN] pri[MN] = {}; //EDF scheduling priorities
17
18 //array for original dependencies, 1 for dependency , 0 for no
       dependency
19 // - {{0,1},{0,0}} means that 1 is dependant on 2
20 const bool origdep[MN][MN]={};
21
22 //dynamically updated array for current dependencies
23 bool depend[MN][MN]={};
24
25 //dynamically updated priorities
26 const int NRSteps=, NROffSteps=, MAXOffStep=, MAXStep=, MAXListSize
       =;
27
28 const int[0,MAXOffStep] OffSteps[NROffSteps] = {};
29 const int[1,MN] OffPrios[NROffSteps][MN] = {};
30
31 const int[0,MAXStep] Steps[NRSteps] = {};
32 const int[1,MN] Prios[NRSteps][MN] = {};
33
34
35 //System-Independent Decl
36 //Synchronization channels
37 chan ready[M], run[M], preempt[M], finish[M], synchronize[M],
       schedule[M];
38 broadcast chan reschedule; //broadcast channel for rescheldulingg
       after a task has finished
```

```
39 broadcast chan go; // broadcast channel for ensuring reaction on
       all signals before scheduling
40 broadcast chan check; // check for change in dynamically updated
       scheduling criteria
41
42 int[0,MN] tauid[M]; //transfer of local taskid from task to
       controller
43 int[0,MN] curtid[M]; //variable used to hold the id of the task
       currently chosen
44 int[0,MN] ltid[M]; //variable used to hold the id of the task
       currently running
45 bool Released[MN]; //array of tasks which have issued ready signals
46 bool Enabled[MN]; //array of tasks which are not awaiting
       dependencies to be resolved
47 bool rescheduleNeeded; //indicator for the need for a global
       reschedule
48 bool Finished[MN]; //array specifying which tasks are finished now
49 bool WaitDep[MN]; //array for tasks which are awaiting dependencies
       to be resolved
50 bool running[M]; //indicating wether a task is currently running on
       the processor
51 bool missedDeadline; //indicator for a missed deadline
52
53 //Locking mechanisms
54 bool Plock[M]; //locking mechanism for the processors ensuring "
       correct" scheduling
55 bool Hold[MN]; //locking mechanism for the tasks ensuring "correct"
       scheduling
56
57 //function checking for dependencies for task t
58 bool taskHasDependency(int t) {
59   for (ini : int[0,MN-1]) {
60     if (depend[t][ini]) {
61       return true;
62     }
63   }
64   return false;
65 }
66
67 //function updating dependencies when task t has finished
68 void opdDep(int t) {
69   for (ini : int[0,MN-1]) {
70     depend[ini][t]=false;
71   }
72 }
73
74
75 //function checking for existance of true boolean value in array of
       size M
76 bool existsPE(bool la[M]) {
77   for (ini : int[0,M-1]) {
78     if (la[ini]) {
79       return true;
80     }
81   }
```

```
82    return false;
83  }
84
85  //function checking for existance of true boolean value in array of
        size MN
86  bool existsT(bool pen[MN]) {
87    bool b = false;
88    for (ini : int[0,MN-1]) {
89      if (pen[ini] == true) {
90        return true;
91      }
92    }
93    return false;
94  }
95
96  //function checking if task t is on processing element p
97  bool isOnPE(int t, int p){
98    for(i:int[0,N-1]){
99      if(onPE[p-1][i]==t) return true;
100   }
101   return false;
102 }
103
104 //function checking if all boolean values in array of size MN are
      true
105 bool allT(bool arr[MN]){
106   for(ini:int[0,MN-1]){
107     if(!arr[ini]) return false;
108   }
109   return true;
110 }
```

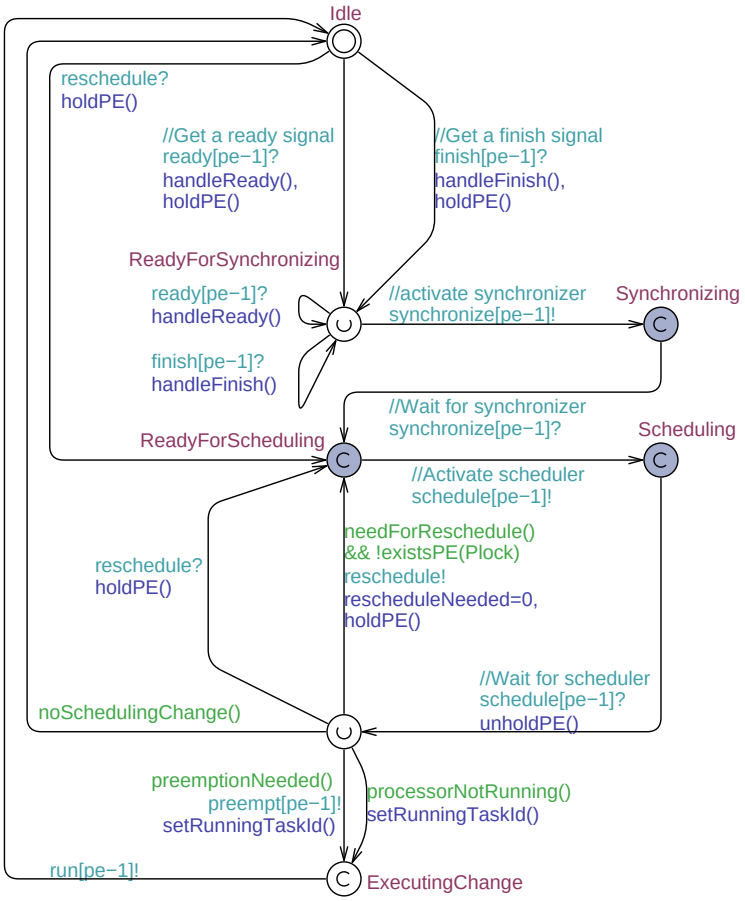
A.2.2 Task template



A.2.3 Task template declarations

```
1  clock cp, cr;
2  int[bcet,wcet] e=bcet;
```

A.2.4 Controller template

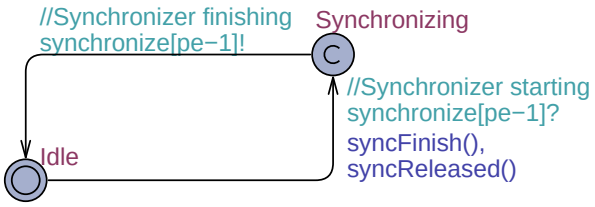


A.2.5 Controller template declarations

```
1  bool processorNotRunning() {
2    return ((!running[pe-1] && curtid[pe-1] != 0) && !
         rescheduleNeeded) && !existsPE(Plock);
3  }
4
5  void holdPE() {
6      Plock[pe-1]=1;
7  }
8
9  void unholdPE() {
```

```
10      Plock[pe-1]=0;
11  }
12
13  void setRunningTaskId() {
14    tauid[pe-1]=curtid[pe-1];
15    ltid[pe-1] = curtid[pe-1];
16    running[pe-1]=1;
17  }
18
19  bool noSchedulingChange() {
20    return (((ltid[pe-1]==curtid[pe-1] && running[pe-1]) || curtid[pe
          -1]==0) && !rescheduleNeeded) && !existsPE(Plock);
21  }
22
23  bool preemptionNeeded() {
24    return ((ltid[pe-1]!=curtid[pe-1] && running[pe-1]) && !
          rescheduleNeeded) && !existsPE(Plock);
25  }
26
27  bool needForReschedule() {
28      return (!existsPE(Plock) && rescheduleNeeded);
29  }
30
31  void setOrigDep(int t) {
32    for (ini : int[0,MN-1]) {
33      if ((offset[t]>offset[ini])&&Finished[ini])
34        depend[t][ini]=false;
35      else
36        depend[t][ini]=origdep[t][ini];
37    }
38  }
39
40  void handleReady(){
41    Finished[tauid[pe-1]-1]=0;
42    Released[tauid[pe-1]-1]=1;
43    setOrigDep(tauid[pe-1]-1);
44  }
45
46  void handleFinish(){
47    Finished[tauid[pe-1]-1]=1;
48    opdDep(tauid[pe-1]-1);
49    Enabled[tauid[pe-1]-1]=0;
50    running[pe-1]=0;
51  }
```

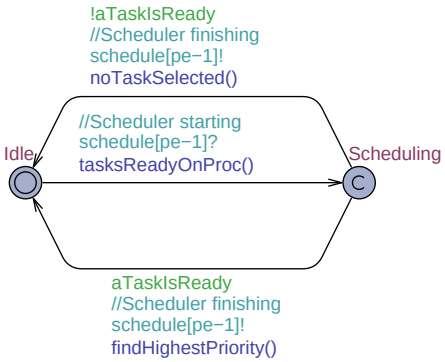
A.2.6 Synchronizer template



A.2.7 Synchronizer template declarations

```
1  void syncFinish() {
2    for (i : int[0,MN-1]) {
3      if (WaitDep[i] && !taskHasDependency(i)) {
4        Enabled[i]=true;
5        WaitDep[i]=false;
6        rescheduleNeeded=true;
7      }
8    }
9  }
10
11 void syncReleased() {
12   for (i : int[0, MN-1]) {
13     if (Released[i] && isOnPE(i+1,pe)) {
14       Released[i]=false;
15       if (taskHasDependency(i)) {
16         WaitDep[i]=true;
17       }
18       else {
19         Enabled[i]=true;
20       }
21     }
22   }
23 }
```

A.2.8 Scheduler template

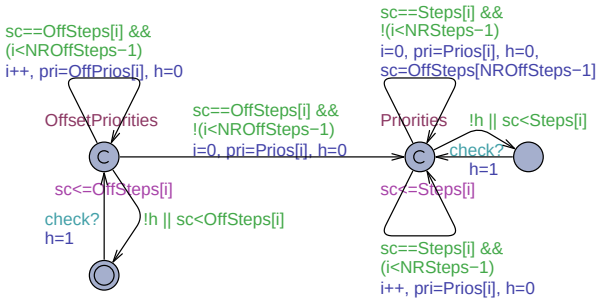


A.2.9 Scheduler template declarations

```
1  int lcri, lcri2; //variables used to hold the criterion of the task
       currently chosen
2  bool aTaskIsReady;
3
4  void noTaskSelected() {
5    curtid[pe-1]=0;
6  }
7
8  void findHighestPriority() {
9    for (i : int[0,MN-1]) {
10     if (isOnPE(i+1,pe)) {
11       if (Enabled[i]) {
12         if (processorScheduling[pe-1] == FP && (fpris[i] < lcri)) {
13           curtid[pe-1]=i+1;
14           lcri=fpris[i];
15         }
16         if (processorScheduling[pe-1] == RM && (pi[i] < lcri || (pi
               [i] == lcri && fpris[i] < lcri2))) {
17           curtid[pe-1]=i+1;
18           lcri=pi[i];
19           lcri2=fpris[i];
20         }
21         if (processorScheduling[pe-1] == EDF && (pri[i] < lcri || (
               pri[i] == lcri && fpris[i] < lcri2))) {
22           curtid[pe-1]=i+1;
23           lcri=pri[i];
24           lcri2=fpris[i];
25         }
26       }
27     }
28   }
29 }
```

```
30
31 void tasksReadyOnProc() {
32   aTaskIsReady = false;
33   for (i : int[0,MN-1]) {
34     if (isOnPE(i+1,pe)) {
35       if (Enabled[i]) {
36         aTaskIsReady = true;
37           if (processorScheduling[pe-1] == FP) {
38             lcri = fpris[i];
39           }
40           if (processorScheduling[pe-1] == RM) {
41             lcri = pi[i];
42             lcri2 = fpris[i];
43           }
44           if (processorScheduling[pe-1] == EDF) {
45             lcri = pri[i];
46             lcri2 = fpris[i];
47           }
48           curtid[pe-1]=i+1;
49           return;
50       }
51     }
52   }
53 }
```

A.2.10 Administrating template



A.2.11 Administrating template declarations

```
1 clock sc; //clock for updating dynamically updated priorities
2 int[0, MAXListSize-1] i=0;
3 bool h;
```

A.2.12 System instantiation

```
1
2  //System-Dependent Inst
3  //Control(pe)
4  //Synchronizer(pe)
5  //Scheduler(pe)
6
7  //Task(pe,tau,bcet,wcet)
8
9  //system Tasks, Cons, Syns, Schs, Tadm;
10
11 //System-Independent Inst
```

A.3 Model with discretization of the running time

This is the full definition of the timed-automata verification structure where the running time was discretized. The task model was presented in Section 5.1.5.

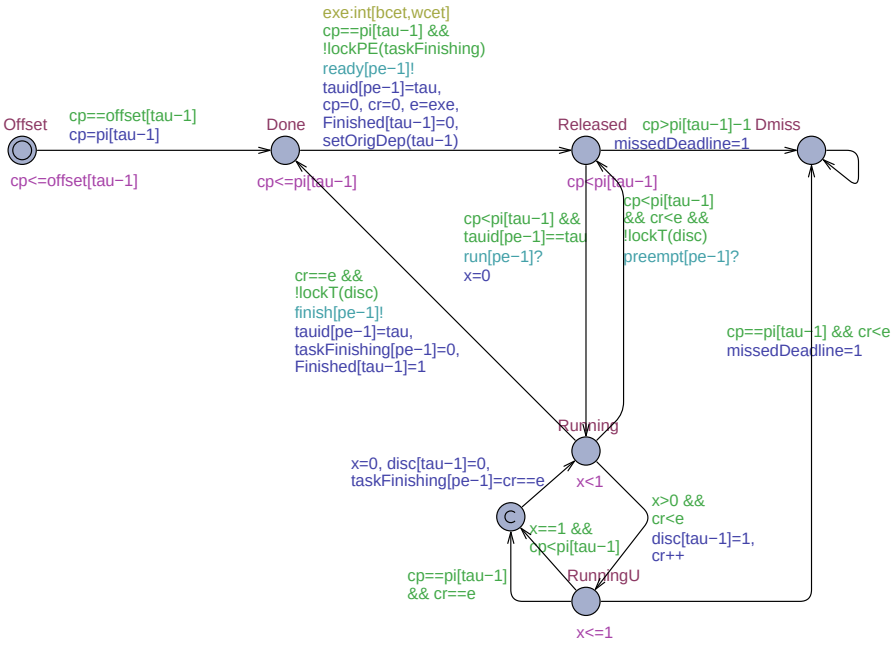
A.3.1 Global declarations

```
1  clock TM;
2
3  const int FP = 0, RM = 1, EDF = 2; //symbolic representation of
       scheduling principles
4
5
6
7  //System-Dependent Decl
8  const int M = ;
9  const int N = ;
10 const int MN = ;
11
12 const int[FP,EDF] processorScheduling[M] = {};
13 const int[0,MN] onPE[M][N] = {};
14 const int fpris[MN] = {};
15 const int pi[MN] = {};
16 const int offset[MN] = {};
17
18 const bool origdep[MN][MN] = {};
19
20 bool depend[MN][MN] = {};
21
22 int[1,MN] pri[MN] = {}; //EDF scheduling priorities
23
24 const int NRSteps=, NROffSteps=, MAXOffStep=, MAXStep=, MAXListSize
       =;
25
26 const int[0,MAXOffStep] OffSteps[NROffSteps] = {};
27 const int[1,MN] OffPrios[NROffSteps][MN] = {};
28
29 const int[0,MAXStep] Steps[NRSteps] = {};
30 const int[1,MN] Prios[NRSteps][MN] = {};
31
32 const int MaxExe=;
33 const int MaxPi=;
34
35 //System-Independent Decl
36 //Synchronization channels
37 broadcast chan reschedule; //broadcast channel for rescheldulingg
       after a task has finished
38 chan synchronize[M], schedule[M];
39 chan ready[M], run[M], preempt[M];
```

```
40 urgent chan finish[M];
41
42 int[0,MN] tauid[M]; //transfer of local taskid from task to
       controller
43 int[0,MN] curtid[M]; //variable used to hold the id of the task
       currently chosen
44 int[0,MN] ltid[M]; //variable used to hold the id of the task
       currently running
45 bool Released[MN]; //array of tasks which have issued ready signals
46 bool Enabled[MN]; //array of tasks which are not awaiting
       dependencies to be resolved
47 bool rescheduleNeeded; //indicator for the need for a global
       reschedule
48 bool Finished[MN]; //array specifying which tasks are finished now
49 bool WaitDep[MN]; //array for tasks which are awaiting dependencies
       to be resolved
50 bool running[M]; //indicating wether a task is currently running on
       the processor
51 bool missedDeadline = false; //indicator for a missed deadline
52
53
54 //Locking mechanisms
55 bool disc[MN]; //locking mechanism for ensuring full discretization
       run
56 bool taskFinishing[M]; //locking mechanism for ensuring reaction to
       all finish signals before any ready signals
57
58 //function checking for dependencies for task t
59 bool taskHasDependency(int t) {
60   for (ini : int[0,MN-1]) {
61     if (depend[t][ini]) {
62       return true;
63     }
64   }
65   return false;
66 }
67
68 //function updating dependencies when task t has finished
69 void opdDep(int t) {
70   for (ini : int[0,MN-1]) {
71     depend[ini][t]=false;
72   }
73 }
74
75
76 //function checking for existance of boolean value in array of size
       M
77 bool lockPE(bool la[M]) {
78   for (ini : int[0,M-1]) {
79     if (la[ini]) {
80       return true;
81     }
82   }
83   return false;
84 }
```

```
85
86  //function checking for existance of boolean value in array of size
        MN
87  bool lockT(bool pen[MN]) {
88    bool b = false;
89    for (ini : int[0,MN-1]) {
90      if (pen[ini] == true) {
91        return true;
92      }
93    }
94    return false;
95  }
96
97  //function checking if task t is on processing element p
98  bool isOnPE(int t, int p){
99    for(i:int[0,N-1]){
100     if(onPE[p-1][i]==t) return true;
101   }
102   return false;
103 }
```

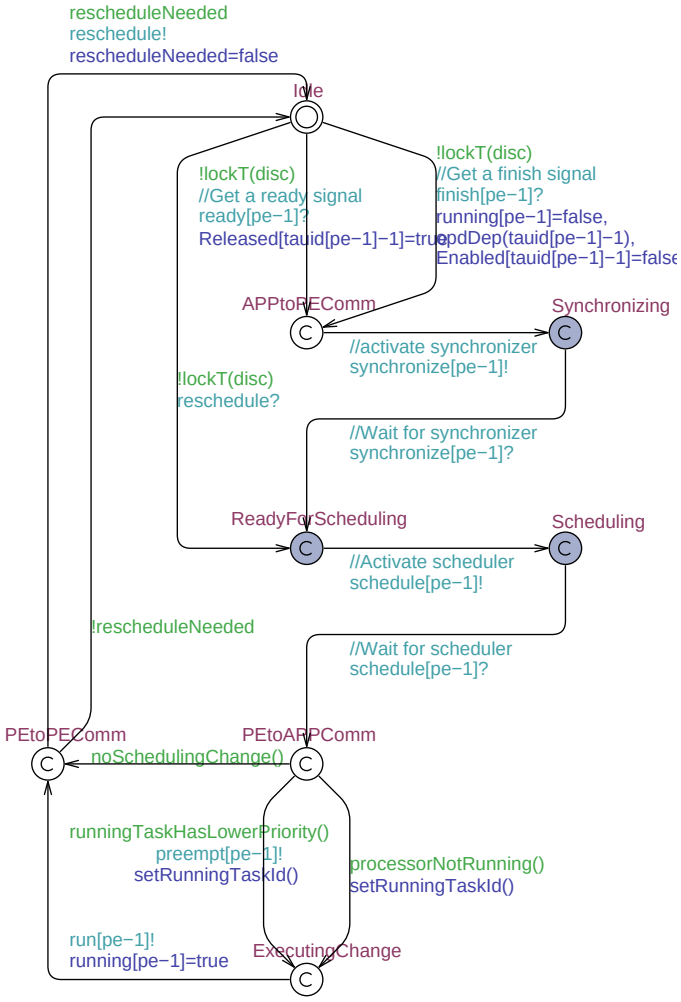
A.3.2 Task template



A.3.3 Task template declarations

```
1  clock cp, x;
2  int[0,wcet] cr;
3  int[bcet,wcet] e=bcet;
4
5  void setOrigDep(int t) {
6    for (ini : int[0,MN-1]) {
7      if ((offset[t]>offset[ini])&&Finished[ini])
8        depend[t][ini]=false;
9      else
10       depend[t][ini]=origdep[t][ini];
11   }
12 }
```

A.3.4 Controller template

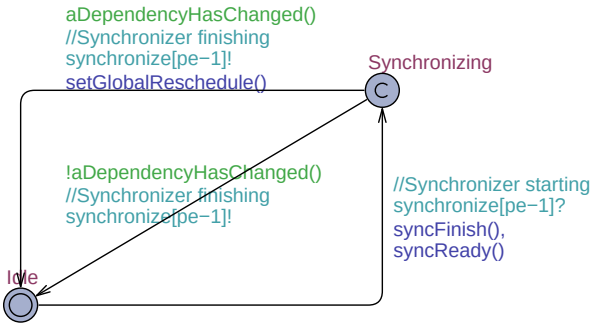


A.3.5 Controller template declarations

```
1  bool processorNotRunning() {
2    return (!running[pe-1] && curtid[pe-1] != 0);
3  }
4
5  void setRunningTaskId() {
6    tauid[pe-1]=curtid[pe-1];
7    ltid[pe-1] = curtid[pe-1];
```

```
8  }
9
10 bool noSchedulingChange() {
11   return ((ltid[pe-1]==curtid[pe-1] && running[pe-1]) || curtid[pe
       -1]==0);
12 }
13
14 bool runningTaskHasLowerPriority() {
15   return (ltid[pe-1]!=curtid[pe-1] && running[pe-1]);
16 }
```

A.3.6 Synchronizer template

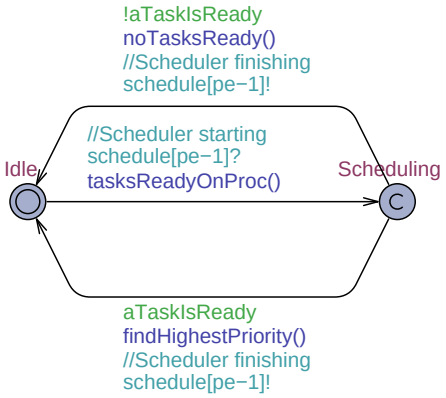


A.3.7 Synchronizer template declarations

```
1  bool depCh; //flag used if a dependency has been changed
2
3
4  bool aDependencyHasChanged() {
5    return (depCh);
6  }
7
8  void setGlobalReschedule() {
9    depCh=false;
10   rescheduleNeeded = true;
11 }
12
13
14
15 void syncFinish() {
16   for (i : int[0,MN-1]) {
17     if (WaitDep[i] && !taskHasDependency(i)) {
18       Enabled[i]=true;
19       WaitDep[i]=false;
20       depCh=true;
21     }
```

```
22    }
23  }
24
25  void syncReady() {
26    for (i : int[0, MN-1]) {
27      if (Released[i] && isOnPE(i+1,pe)) {
28        Released[i]=false;
29        if (taskHasDependency(i)) {
30          WaitDep[i]=true;
31        }
32        else {
33          Enabled[i]=true;
34        }
35      }
36    }
37  }
```

A.3.8 Scheduler template

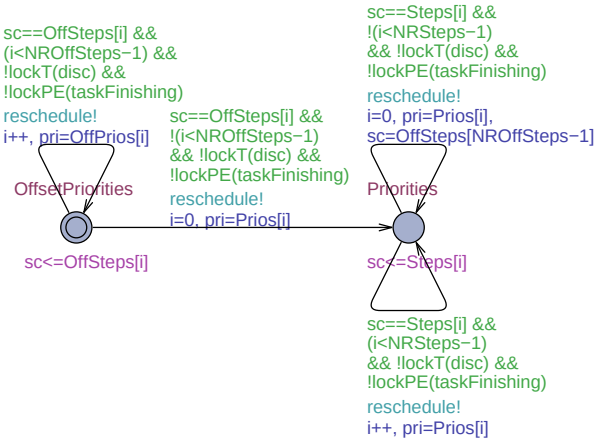


A.3.9 Scheduler template declarations

```
1   int[0,MN] lcri, lcri2; //variables used to hold the criterion of
        the task currently chosen
2   int[0,MaxPi] lcrirm;
3   bool aTaskIsReady;
4
5   void noTasksReady() {
6     curtid[pe-1]=0;
7   }
8
9   void findHighestPriority() {
10    for (i : int[0,MN-1]) {
11      if (isOnPE(i+1,pe)) {
```

```
12        if (Enabled[i]) {
13          if (processorScheduling[pe-1] == FP && (fpris[i] < lcri)) {
14            curtid[pe-1]=i+1;
15            lcri=fpris[i];
16          }
17          if (processorScheduling[pe-1] == RM && (pi[i] < lcrirm || (
                pi[i] == lcrirm && fpris[i] < lcri2))) {
18            curtid[pe-1]=i+1;
19            lcrirm=pi[i];
20            lcri2=fpris[i];
21          }
22          if (processorScheduling[pe-1] == EDF && (pri[i] < lcri || (
                pri[i] == lcri && fpris[i] < lcri2))) {
23            curtid[pe-1]=i+1;
24            lcri=pri[i];
25            lcri2=fpris[i];
26          }
27        }
28      }
29    }
30  }
31
32  void tasksReadyOnProc() {
33    aTaskIsReady = false;
34    for (i : int[0,MN-1]) {
35      if (isOnPE(i+1,pe)) {
36        if (Enabled[i]) {
37          aTaskIsReady = true;
38            if (processorScheduling[pe-1] == FP) {
39              lcri = fpris[i];
40            }
41            if (processorScheduling[pe-1] == RM) {
42              lcrirm = pi[i];
43              lcri2 = fpris[i];
44            }
45            if (processorScheduling[pe-1] == EDF) {
46              lcri = pri[i];
47              lcri2 = fpris[i];
48            }
49            curtid[pe-1]=i+1;
50            return;
51        }
52      }
53    }
54  }
```

A.3.10 Administrating template



A.3.11 Administrating template declarations

```
1  clock sc; //clock for updating dynamically updated priorities
2  int[0,MAXListSize-1] i=0;
```

A.3.12 System instantiation

```
1
2  //System-Dependent Inst
3  //Control(pe)
4  //Synchronizer(pe)
5  //Scheduler(pe)
6
7  //Task(pe,tau,bcet,wcet)
8
9  //system Tasks, Cons, Syns, Schs, Tadm;
10
11 //System-Independent Inst
```

A.4 Genuine discrete model

This is the full definition of the genuine discrete verification structure. The task model was presented in Section 5.1.5.

A.4.1 Global declarations

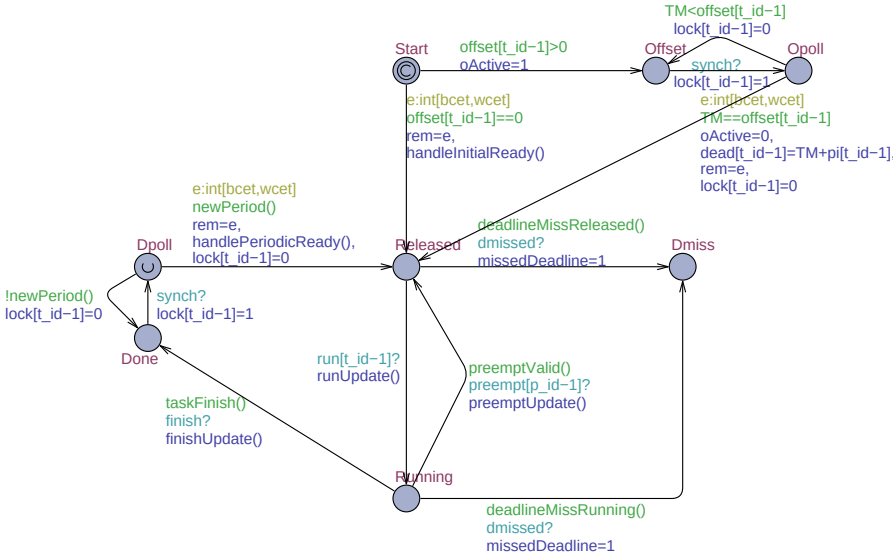
```
1  //Constants for scheduling principles
2  const int FP=0, RM=1, EDF=2;
3
4  //System-Dependent Decl
5  const int M = ; //The number of Processors
6  const int N = ; //The maximum number of tasks per Processor
7  const int MN = ; //The total number of tasks
8
9  const int[FP,EDF] processorScheduling[M] = {};
10 const int[0,MN] onPE[M][N] = {};
11 const int fpris[MN] = {};
12 const int pi[MN] = {};
13 const int offset[MN] = {};
14
15 const bool origdep[MN][MN] = {};
16
17 bool depend[MN][MN] = {};
18
19 int[1,MN] pri[MN] = {}; //EDF scheduling priorities
20
21 const int NRSteps=, NROffSteps=, MAXOffStep=, MAXStep=;
22
23 const int[0,MAXOffStep] OffSteps[NROffSteps] = {};
24 const int[1,MN] OffPrios[NROffSteps][MN] = {};
25
26 const int[0,MAXStep] Steps[NRSteps] = {};
27 const int[1,MN] Prios[NRSteps][MN] = {};
28
29 const int MaxExe=;
30 const int MaxPi=;
31
32 //System-Independent Decl
33 int[0,MAXStep] TM;
34 int[0,MAXStep] cfin[M];
35 bool cfinNextPer[M];
36 bool cfinValid[M];
37 bool missedDeadline;
38
39 broadcast chan synch, finish, dmissed, prc;
40 chan run[MN], preempt[M];
41 //bool released[MN];
42 int[0,MAXStep] dead[MN];
43 bool deadNextPer[MN];
44 int[0,MAXStep] st[M];
```

```
45 int[0,MN] executing[M];
46 bool lock[MN], lockp[M];
47
48 bool Released[MN]; //array of tasks which have issued ready signals
49 bool Enabled[MN]; //array of tasks which are not awaiting
       dependencies to be resolved
50 bool WaitDep[MN]; //array for tasks which are awaiting dependencies
       to be resolved
51 bool rescheduleNeeded; //indicator for the need for a global
       reschedule
52 bool Finished[MN];
53
54 void setOrigDep(int t) {
55   for (ini : int[0,MN-1]) {
56     if ((offset[t]>offset[ini])&&Finished[ini])
57       depend[t][ini]=false;
58     else
59       depend[t][ini]=origdep[t][ini];
60   }
61 }
62
63 //function updating dependencies when task t has finished
64 void opdDep(int t) {
65   for (ini : int[0,MN-1]) {
66     depend[ini][t]=false;
67   }
68 }
69
70 //function checking if task t is on processing element p
71 bool isOnPE(int t, int p){
72   for(i:int[0,N-1]){
73     if(onPE[p-1][i]==t) return true;
74   }
75   return false;
76 }
77
78 bool anyT(bool array[MN])
79 {
80   for(i:int[0,MN-1])
81   {
82     if(array[i])
83     {
84       return 1;
85     }
86   }
87   return 0;
88 }
89
90 bool anyP(bool array[M])
91 {
92   for(i:int[0,M-1])
93   {
94     if(array[i])
95     {
96       return 1;
```

```
97      }
98    }
99    return 0;
100 }
101
102 int minCfin(int stp)
103 {
104   bool vld;
105   int[0,MAXStep*2] ret;// = cfin[0]+(cfinNextPer[0]*MAXStep);
106   for(i:int[0,M-1])
107   {
108     if(!vld && cfinValid[i])
109     {
110       ret=cfin[i]+(cfinNextPer[i]*MAXStep);
111       vld=1;
112     }
113     if(cfin[i]+(cfinNextPer[i]*MAXStep)<ret && cfinValid[i])
114     {
115       ret=cfin[i]+(cfinNextPer[i]*MAXStep);
116     }
117   }
118   if(!vld)
119   {
120     return stp+1;
121   }
122   else
123   {
124     return ret;
125   }
126 }
127
128
129 int nextEventTime(int nextStep)
130 {
131   int[0,MAXStep] res = nextStep;
132   for(i:int[0,M-1])
133   {
134     if(cfin[i]<res && cfinValid[i] && !cfinNextPer[i])
135     {
136       res=cfin[i];
137     }
138   }
139   return res;
140 }
141
142 bool dmissCheck(int timep)
143 {
144   for(i:int[0,MN-1])
145   {
146     if(dead[i]<timep)
147     {
148       return 1;
149     }
150   }
151   return 0;
```

```
}
```

A.4.2 Task template

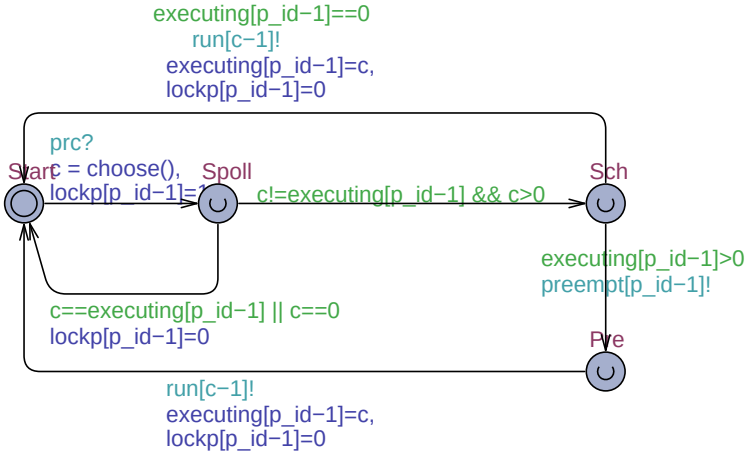


A.4.3 Task template declarations

```
1  // Place local declarations here.
2  int[0,MaxExe] rem;
3  bool oActive;
4
5  void handleReady(){
6    Finished[t_id-1]=0;
7    Released[t_id-1]=1;
8    setOrigDep(t_id-1);
9  }
10
11 void handlePeriodicReady(){
12   dead[t_id-1]=TM+pi[t_id-1]>MAXStep?TM+pi[t_id-1]-MAXStep+
         MAXOffStep:TM+pi[t_id-1];
13   deadNextPer[t_id-1]=TM+pi[t_id-1]>MAXStep?1:0;
14   handleReady();
15 }
16
17 void handleInitialReady(){
18   dead[t_id-1]=TM+pi[t_id-1];
19   handleReady();
20 }
```

```
21
22 void handleFinish(){
23   Finished[t_id-1]=1;
24   opdDep(t_id-1);
25   Enabled[t_id-1]=0;
26 }
27
28 bool deadlineMissReleased(){
29   return TM>=dead[t_id-1]+deadNextPer[t_id-1]*(MAXStep-MAXOffStep);
30 }
31
32 bool deadlineMissRunning(){
33   return dead[t_id-1]+(MAXStep-MAXOffStep)*deadNextPer[t_id-1] <
         cfin[p_id-1]+(MAXStep-MAXOffStep)*cfinNextPer[p_id-1];
34 }
35
36 bool taskFinish(){
37   return TM<=deadNextPer[t_id-1]*(MAXStep-MAXOffStep)+dead[t_id-1]
       && TM==cfinNextPer[p_id-1]*(MAXStep-MAXOffStep)+cfin[p_id-1];
38 }
39
40 bool preemptValid(){
41   return TM<dead[t_id-1]+deadNextPer[t_id-1]*(MAXStep-MAXOffStep);
42 }
43
44 bool newPeriod(){
45   return (TM-offset[t_id-1]) % pi[t_id-1]==0;
46 }
47
48 void preemptUpdate(){
49   rem=rem-(TM-st[p_id-1]);
50   cfin[p_id-1]=0;
51   cfinValid[p_id-1]=0;
52 }
53
54 void runUpdate(){
55   st[p_id-1]=TM;
56   cfin[p_id-1]=TM+rem>MAXStep?TM+rem-MAXStep+MAXOffStep:TM+rem;
57   cfinNextPer[p_id-1]=TM+rem>MAXStep?1:0;
58   cfinValid[p_id-1]=1;
59 }
60
61 void finishUpdate(){
62   cfin[p_id-1]=0;
63   executing[p_id-1]=0;
64   cfinValid[p_id-1]=0;
65   handleFinish();
66 }
```

A.4.4 Controller template



A.4.5 Controller template declarations

```
1  int[0,MN] c;
2  int choose()
3  {
4    int[0,MN] cand=0;
5    int[0,MN] lpri;
6    int[0,MaxPi] rmlpri;
7    for(i:int[0,MN-1])
8    {
9      if(Enabled[i] && isOnPE(i+1,p_id))
10     {
11       if(cand==0)
12       {
13         if(processorScheduling[p_id-1]==EDF)
14         {
15           lpri=pri[i];
16         }
17         if(processorScheduling[p_id-1]==RM)
18         {
19           rmlpri=pi[i];
20         }
21         if(processorScheduling[p_id-1]==FP)
22         {
23           lpri=fpris[i];
24         }
25         cand=i+1;
26       }
27       else
28       {
```

```
29          if(processorScheduling[p_id-1]==EDF)
30          {
31            if(pri[i]<lpri)
32            {
33              lpri=pri[i];
34              cand=i+1;
35            }
36          }
37          if(processorScheduling[p_id-1]==RM)
38          {
39            if(pi[i]<rmlpri)
40            {
41              rmlpri=pi[i];
42              cand=i+1;
43            }
44          }
45          if(processorScheduling[p_id-1]==FP)
46          {
47            if(fpris[i]<lpri)
48            {
49              lpri=fpris[i];
50              cand=i+1;
51            }
52          }
53        }
54      }
55    }
56    return cand;
57  }
```

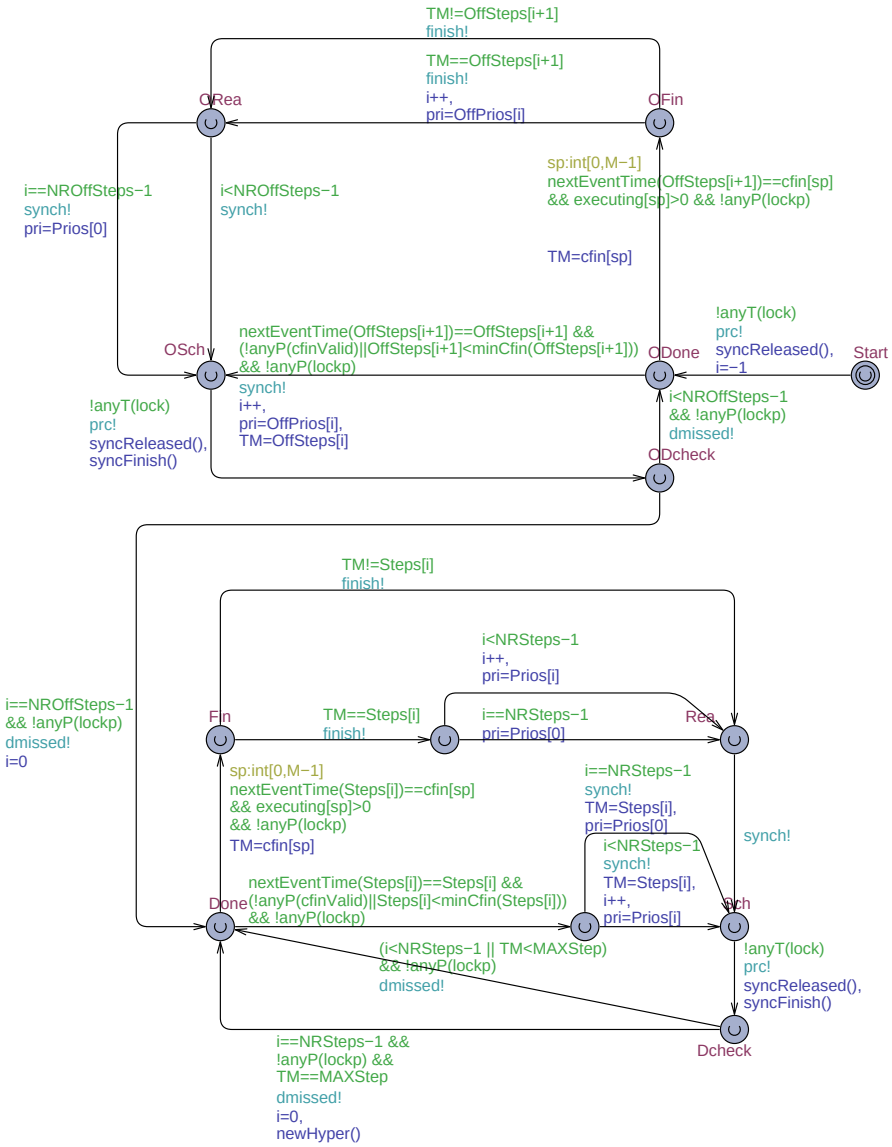
A.4.6 Synchronizer template

A.4.7 Synchronizer template declarations

A.4.8 Scheduler template

A.4.9 Scheduler template declarations

A.4.10 Administrating template



A.4.11 Administrating template declarations

```
1  int[-1,NRSteps] i = 0;
2
3  void newHyper()
4  {
5    for(i:int[0,MN-1])
6    {
7      if(deadNextPer[i])
8      {
9        deadNextPer[i]=0;
10 //       dead[i]=dead[i]-(MAXStep-MAXOffStep);
11     }
12   }
13   for(i:int[0,M-1])
14   {
15     if(cfinNextPer[i])
16     {
17       cfinNextPer[i]=0;
18       st[i]=st[i]-MAXStep+MAXOffStep;
19     }
20   }
21
22 }
23
24 //function checking for dependencies for task t
25 bool taskHasDependency(int t) {
26   for (ini : int[0,MN-1]) {
27     if (depend[t][ini]) {
28       return true;
29     }
30   }
31   return false;
32 }
33
34 void syncReleased() {
35   for (i : int[0, MN-1]) {
36     if (Released[i]) {
37       Released[i]=false;
38       if (taskHasDependency(i)) {
39         WaitDep[i]=true;
40       }
41       else {
42         Enabled[i]=true;
43       }
44     }
45   }
46 }
47
48 void syncFinish() {
49   for (i : int[0,MN-1]) {
50     if (WaitDep[i] && !taskHasDependency(i)) {
51       Enabled[i]=true;
52       WaitDep[i]=false;
53       rescheduleNeeded=true;
```

```
54      }
55    }
56  }
```

A.4.12 System instantiation

```
1
2   //System-Dependent Inst
3   //Control(pe)
4   //Synchronizer(pe)
5   //Scheduler(pe)
6
7   //Task(pe,tau,bcet,wcet)
8
9   //system Tasks, Cons, Syns, Schs, Tadm;
10
11  //System-Independent Inst
```

Appendix B

Source Code for the MoVES Tool

This appendix includes the full source code for the MoVES tool. There are lexer and parser definitions for both the frontend and the trace generator. Furthermore, all the SML functions that make up the MoVES tool are given here.

B.1 Frontend

Here we give the abstract syntax used for the frontend and the model generator. Also, there is lexer and parser definitions for the frontend and a few auxiliary functions used for parsing.

B.1.1 Abstract syntax

```
1  (* Absyn.sml: Abstract syntax for MoVES
2     Aske Brekling 13/5/2008
3   *)
4
5  exception noSuchSchPrinciple
6  exception noSuchArbiter
```

```
7  exception noSuchT
8  exception emptyMapping
9  exception mappingError
10 type tname = string
11 type pname = string
12 type bname = string
13 type bcet = int
14 type wcet = int
15 type period = int
16 type offset = int
17 type speed = int
18 type data = int
19 datatype sch = FP | RM | EDF
20 datatype arb = FIFO
21 datatype task = T of tname * bcet * wcet * period * offset
22 datatype mtask = MT of tname * period * offset
23 datatype pe = P of pname * sch
24 datatype bus = B of bname * arb * speed
25 type appl = task list
26 type plat = pe list
27 type mplat = pe list * bus
28 type mapping = (tname * pname) list
29 type depend = (tname * tname) list
30 type mdepend = (tname * tname * data) list
31 datatype system = S of (plat * appl * mapping * depend)
32 type tonp = tname * pname * bcet * wcet
33 type chr = tonp list
34 type mappl = mtask list * mdepend
35 datatype prop = Schedule | Trace
36 datatype msys = M of (mplat * mappl * (tname * pname) list * chr *
       prop)
37
38 fun mkSch s = case s of
39       "FP" => FP
40     | "RM" => RM
41     | "EDF" => EDF
42     | _ => raise noSuchSchPrinciple
43
44 fun mkArb a = case a of
45       "FIFO" => FIFO
46     | _ => raise noSuchArbiter
47
48 fun mkMp [] = raise emptyMapping
49     | mkMp((t1,p1)::tps) = foldl (fn((t',p'),g) => fn(t1) => (if t1=t
         ' then p' else (g(t1)))) (fn t => (if t=t1 then p1 else raise
          noSuchT)) ((t1,p1)::tps)
```

B.1.2 Lexer (lex)

```
1  {
2  (* MoVESlex.lex: lexer specification for MoVES
3     Aske Brekling 13/05/2008
4   *)
```

```
5
6  open Lexing MoVESpar;
7
8  exception LexicalError of string * int * int (* (message, loc1,
       loc2) *)
9
10 fun lexerError lexbuf s =
11     raise LexicalError (s, getLexemeStart lexbuf, getLexemeEnd
           lexbuf);
12
13 }
14
15 rule Token = parse
16     [' ' '\t' '\n' '\r']  { Token lexbuf }
17   | "Application"         { APP }
18   | "Platform"            { PLAT }
19   | "Task:"               { TASK }
20   | "Bcet:"               { BCET }
21   | "Wcet:"               { WCET }
22   | "Period:"             { PERIOD }
23   | "Offset:"             { OFFSET }
24   | "Proc:"               { PE }
25   | "Sch:"                { SCH }
26   | "Bus:"                { BUS }
27   | "Arb:"                { ARB }
28   | "Mapping"             { MAP }
29   | "Dependencies"        { DEP }
30   | "Characteristics"     { CHR }
31   | "Speed:"              { SPEED }
32   | "@"                   { AT }
33   | ":"                   { COLON }
34   | "->"                  { PREC }
35   | "Property"            { PROP }
36   | "Schedule?"           { SCHEDULE }
37   | "Trace!"              { TRACE }
38
39   | ['0'-'9']+            { case Int.fromString (getLexeme lexbuf)
       of
40                                 NONE   => lexerError lexbuf "
                                       internal error"
41                               | SOME i => INT i
42                           }
43   | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9']*
44                           { NAME (getLexeme lexbuf) }
45   | eof                   { EOF   }
46   | _                     { lexerError lexbuf "Illegal symbol in
       input" }
47 and SkipToEndLine = parse
48     ['\n' '\r']           { () }
49    | (eof | '\^Z')        { () }
50    | _                    { SkipToEndLine lexbuf }
51 ;
```

B.1.3 Parser (yacc)

```
%{
(* MoVESpar.grm: parser specification for MoVES
   Aske Brekling 13/5/2008
 *)

  open Absyn;
%}

%token <int> INT                          // Accepting numbers
%token <string> NAME                      // Accepting names
%token APP PLAT MAP DEP ARB CHR           // Tokens for system
    keywords
%token TASK BCET WCET PERIOD OFFSET       // Tokens for application
    keywords
%token PE SCH COLON PREC AT SPEED BUS     // Tokens for more keywords
%token PROP SCHEDULE TRACE                // Tokens for more keywords
%token EOF                                // Token for end of file


%start Main

%type <Absyn.msys> Main
%type <Absyn.mappl> app
%type <Absyn.mtask list> tasklist
%type <Absyn.mtask> task
%type <Absyn.mplat> plat
%type <Absyn.pe list> pelist
%type <Absyn.pe> pe
%type <Absyn.bus> bus
%type <(Absyn.tname * Absyn.pname) list> mplist mapping
%type <Absyn.tname * Absyn.pname> mp
%type <Absyn.mdepend> dependencies
%type <Absyn.tname * Absyn.tname * Absyn.data> dep
%type <Absyn.chr> chr tonplist
%type <Absyn.tonp> tonp
%type <Absyn.prop> prop

%%

// The main program
Main:
    app plat mapping chr prop EOF        { M($2,$1,$3,$4,$5) }
;

app:
    APP tasklist DEP dependencies        { ($2,$4) }
;

tasklist:
    task                          { [$1] }
    | task tasklist               { $1 :: $2 }
;

```

```
52  task:
53      TASK NAME PERIOD INT OFFSET INT      { MT($2,$4,$6) }
54  ;
55
56  plat:
57      PLAT pelist bus                      { ($2, $3) }
58  ;
59
60  pelist:
61      pe                   { [$1] }
62      | pe pelist          { $1 :: $2 }
63  ;
64
65  pe:
66      PE NAME SCH NAME     { P($2, mkSch($4)) }
67  ;
68
69  bus:
70      BUS NAME ARB NAME SPEED INT { B($2, mkArb($4), $6) }
71  ;
72
73  mapping:
74      MAP mplist           { $2 }
75  ;
76
77  mplist:
78      mp                   { [$1] }
79      | mp mplist          { ($1::$2) }
80  ;
81
82  mp:
83      NAME COLON NAME      { ($1, $3) }
84  ;
85
86  dependencies:
87      /* empty */          { [] }
88      | dep dependencies   { $1::$2 }
89  ;
90
91  dep:
92      NAME PREC NAME COLON INT     { ($1, $3, $5) }
93  ;
94
95  chr:
96      CHR tonplist         {$2}
97  ;
98
99  tonplist:
100     tonp                 {[$1]}
101   | tonp tonplist        {$1::$2}
102 ;
103
104 tonp:
105     NAME AT NAME BCET INT WCET INT       {($1,$3,$5,$7)}
106 ;
```

```
107
108 prop:
109     PROP SCHEDULE                          { Schedule }
110   | PROP TRACE                             { Trace }
111 ;
```

B.1.4 Auxiliary functions for the frontend

```
1  (* Lexer and parser for MoVES using mosmllex and mosmlyac
2     Aske Brekling 13/5/2008
3   *)
4  open Absyn;
5
6  (* Plain parsing from a string, with poor error reporting *)
7
8  fun parse str =
9      let val lexbuf = Lexing.createLexerString str
10         val expr   = MoVESpar.Main MoVESlex.Token lexbuf
11     in
12         Parsing.clearParser();
13         expr
14     end
15     handle exn => (Parsing.clearParser(); raise exn);
16
17
18 (* Fancy parsing from a file; show the offending program piece on
       error *)
19
20 fun parseExprReport file stream lexbuf =
21     let val expr =
22             MoVESpar.Main MoVESlex.Token lexbuf
23             handle
24                Parsing.ParseError f =>
25                    let val pos1 = Lexing.getLexemeStart lexbuf
26                        val pos2 = Lexing.getLexemeEnd lexbuf
27                    in
28                        Location.errMsg (file, stream, lexbuf)
29                                        (Location.Loc(pos1, pos2))
30                                        "Syntax error."
31                    end
32              | MoVESlex.LexicalError(msg, pos1, pos2) =>
33                    if pos1 >= 0 andalso pos2 >= 0 then
34                        Location.errMsg (file, stream, lexbuf)
35                                        (Location.Loc(pos1, pos2))
36                                        ("Lexical error: " ^ msg)
37                    else
38                        (Location.errPrompt ("Lexical error: " ^ msg
                              ^ "\n\n");
39                         raise Fail "Lexical error");
40     in
41         Parsing.clearParser();
42         expr
43     end
44     handle exn => (Parsing.clearParser(); raise exn);
```

```
45
46  (* Parse a program from a string, with error reporting *)
47
48  fun parses str =
49      parseExprReport "" (BasicIO.std_in) (Lexing.createLexerString
            str);
50
51  (* Create lexer from instream *)
52
53  fun createLexerStream (is : BasicIO.instream) =
54      Lexing.createLexer (fn buff => fn n => Nonstdio.buff_input is
            buff 0 n)
55
56  (* Parse a program from a file, with error reporting *)
57
58  fun parsef file =
59      let val is     = Nonstdio.open_in_bin file
60          val expr   = parseExprReport file is (createLexerStream is)
61                       handle exn => (BasicIO.close_in is; raise exn)
62      in
63          BasicIO.close_in is;
64          expr
65      end;
```

B.2 Model generator

Here are the SML functions that make up the model generator.
Interesting signatures (based on types from the abstract syntax):

```
filename = string
vs_filename = filename (*verification structure*)
input_filename = filename
output_filename = filename
optimize = "U" (*unoptimized lists*) | "O" (*optimized lists*)
symbol = "," | "<" (*delimiter for system instantiation*)
xml_rep = string

readModel: input_filename -> (xml_rep * xml_rep * xml_rep)
(* Splits up the system-independent parts of the verification
   structure in three parts based on tokens; before \\System-
   Dependent Decl, after \\System-Independent Decl and after
   \\System-Dependent Inst. *)

mkDeclText: (system * optimize) -> xml_rep
(* Generates the system-dependent declarations of the
   verifiable impelementation. *)

mkFullSysDecl: (system * symbol) -> xml_rep
(* Generates the system-dependent instantiations of
   the verifiable implementation. *)

collectFullModel: (vs_filename * system * optimize * symbol)
                                                    -> xml_rep
(* Collects the verifiable implementation of the system in a .xml
   representation in the Uppaal syntax for timed automata. *)

transMtoS: msys -> (system * prop)
(* Transforms a representation in msys (close to MoVES syntax)
   to a representation in system (closer to uppaal syntax). *)

mkQuery: (output_filename * input_filename) -> unit
(* Generates a query file. *)

mkModelFromFile: (vs_filename * input_filename *
                  output_filename * optimize * symbol) -> unit
(* Generates a file with an .xml representation of a verifiable
```

```
   timed-automata implementation. *)

infix isin;
open parser;
exception EmptyList;

fun getPName (P(n,_)) = n
fun getSch (P(_,s)) = s
fun getTName (T(n,_,_,_,_)) = n
fun getBC (T(_,b,_,_,_)) = b
fun getWC (T(_,_,w,_,_)) = w
fun getPi (T(_,_,_,p,_)) = p
fun getO (T(_,_,_,_,off)) = off

exception wrongName;
local
    fun isPName(n,p) = n=getPName p
in
    fun getP(_,[]) = raise wrongName
      | getP(n,p::rest) = if isPName(n,p) then p else getP(n,rest)
    fun getPIndex(_,_,[]) = raise wrongName
      | getPIndex(i,n,p::rest) = if isPName(n,p) then i else
          getPIndex(i+1,n,rest)
end

local
    fun isTName(n,t) = n=getTName t
in
    fun getT(_,[]) = raise wrongName
      | getT(n,t::rest) = if isTName(n,t) then t else getT(n,rest)
end

fun remDuplicates(lst) = List.foldr (fn(x,ys) => if (List.exists (
    fn y => x=y) ys) then ys else x::ys) [] lst

fun constructTList [] = []
  | constructTList(t::rest) = getTName t::constructTList(rest)

fun getPforT (_,[]) = raise mappingError
  | getPforT (t, (tn,pn)::rest) = if t=tn
                                  then pn:pname
                                  else getPforT(t,rest)


fun constructPList ([],_) = []
  | constructPList(t::app,mp) = getPforT(getTName t,mp)::
      constructPList(app,mp)

fun nrOfOccur p [] = 0
  | nrOfOccur p (pn::rest) = (if p=pn then 1 else 0) + nrOfOccur p
      rest

fun extractPs [] = []
  | extractPs(p::rest) = getPName p::extractPs(rest)

```

```
50  fun collectAux [] _ = []
51    | collectAux (p::rest) cpl = (nrOfOccur p cpl)::(collectAux rest
        cpl)
52
53  fun collectNrOfOcc(plat,app,mp) = collectAux (extractPs(plat)) (
      constructPList(app,mp))
54
55  fun maxOfIntList [] = 0
56    | maxOfIntList (l::lst) = let val m = maxOfIntList lst
57                              in if l>m then l else m
58                              end
59
60  fun maxTonP(plat,app,mp,dp) = let val (occList) = collectNrOfOcc(
      plat,app,mp)
61                                in maxOfIntList occList
62                                end
63
64  fun mkPSch FP = "FP"
65    | mkPSch RM = "RM"
66    | mkPSch EDF = "EDF"
67
68  fun mkProcSch [] = raise EmptyList
69    | mkProcSch([p]) = mkPSch(getSch p)
70    | mkProcSch(p::rest) = mkPSch(getSch p)^",_"^(mkProcSch rest)
71
72  fun mkOnPET(_,[],_,_) = raise EmptyList
73    | mkOnPET(i,[T(t,_,_,_,_)],p,mp) = if (getPforT(t,mp)=p) then [i]
        else []
74    | mkOnPET(i,T(t,_,_,_,_)::rest,p,mp) = if getPforT(t,mp)=p then i
        ::mkOnPET(i+1,rest,p,mp) else mkOnPET(i+1,rest,p,mp)
75
76  fun mkOnPE(_,[],_) = raise EmptyList
77    | mkOnPE(ts,[p],mp) = [mkOnPET(1,ts,getPName p,mp)]
78    | mkOnPE(ts,p::rest,mp) = (mkOnPET(1,ts,getPName p,mp))::(mkOnPE(
        ts,rest,mp))
79
80  fun mkOnPETAux(_,0) = ""
81    | mkOnPETAux([],1) = "0"
82    | mkOnPETAux([],i) = "0,_"^mkOnPETAux([],i-1)
83    | mkOnPETAux([t],1) = Int.toString(t)
84    | mkOnPETAux([t],i) = Int.toString(t)^",_"^mkOnPETAux([],i-1)
85    | mkOnPETAux(t::rest,i) = Int.toString(t)^",_"^mkOnPETAux(rest,i
        -1)
86
87  fun mkOnPETextAux([],_) = raise EmptyList
88    | mkOnPETextAux([tlst],nr) = "{"^mkOnPETAux(tlst,nr)^"}"
89    | mkOnPETextAux(tlst::rest,nr) = "{"^mkOnPETAux(tlst,nr)^"},"^
        mkOnPETextAux(rest,nr)
90
91  fun mkOnPEText(plat,app,mp,dp) = "{"^mkOnPETextAux(mkOnPE(app,plat,
      mp),maxTonP(plat,app,mp,dp))^"};\n"
92
93  fun mkFPris(_,[]) = raise EmptyList
94    | mkFPris(i,[t]) = Int.toString(i)
95    | mkFPris(i,t::rest) = Int.toString(i)^",_"^mkFPris(i+1,rest)
```

```
96
97  fun appStrLst [] = ""
98    | appStrLst [a] = a
99    | appStrLst (h::t) = h^",␣"^appStrLst t
100
101 fun mkX fnx tskl = appStrLst (map Int.toString (map fnx tskl))
102
103 val mkPi2 = mkX getPi
104
105 val mkO2 = mkX getO
106
107 fun mkPi [] = raise EmptyList
108   | mkPi([t]) = Int.toString(getPi t)
109   | mkPi(t::rest) = Int.toString(getPi t)^",␣"^mkPi(rest)
110
111 fun mkO [] = raise EmptyList
112   | mkO([t]) = Int.toString(getO t)
113   | mkO(t::rest) = Int.toString(getO t)^",␣"^mkO(rest)
114
115 fun _ isin [] = false
116   | a isin (b::rest) = if a=b then true else a isin rest
117
118 fun mkDep(_,[],_) = raise EmptyList
119   | mkDep(t,[tn],dp) = (if ((getTName tn,t) isin dp) then Int.
        toString(1) else Int.toString(0))
120   | mkDep(t,tn::rest,dp) = (if ((getTName tn,t) isin dp) then Int.
        toString(1) else Int.toString(0)) ^ ",␣" ^ mkDep(t,rest,dp)
121
122 fun mkDepRel([],_,_) = ""
123   | mkDepRel([t],ot,dp) = "{"^mkDep(getTName t,ot,dp)^"}"
124   | mkDepRel(t::rest,ot,dp) = "{"^mkDep(getTName t,ot,dp)^"},"^
        mkDepRel(rest,ot,dp)
125
126 fun mkDepend(app,dp) = "{"^mkDepRel(app,app,dp)^"};\n"
127
128 exception MixedLists;
129 local
130   (* findSmallestD: (int * _ * _)list -> int
131    * Gives the smallest relative deadline in the list
132    *)
133   fun findSmallestD [] = raise EmptyList
134     | findSmallestD [(d,_,_)] = d
135     | findSmallestD ((d,_,_)::xs) =
136     let val sm = findSmallestD xs
137     in
138       if d<sm
139       then d
140       else sm
141     end
142
143   local
144     (* timeStep: (int * int * int * 'a) -> (int * int * 'a)
145      * Makes a time step for a single element in the list
146      *)
```

```
147       fun timeStep(dead,step,per,e) = (((dead-step-1) mod per)+1,per,
              e)
148     in
149       (* timeStepList: (int * int * 'a) list -> (int * int * 'a) list
150        * Makes a time step for the whole list with the smallest
               element
151        *)
152       fun timeStepList ds = map (fn (x,y,e) => timeStep(x,
              findSmallestD ds,y,e)) ds
153     end
154     local
155       (* findAndRemoveSmallest: (int * int * 'a) list -> (int * int
              list)
156        * Gives the element in the list with the smallest relative
               deadline as well as the
157        * list without that element
158        *)
159       fun findAndRemoveSmallest [] = raise EmptyList
160         | findAndRemoveSmallest [e] = (e,[])
161         | findAndRemoveSmallest(x::xs) =
162         let val (sm,rest) = findAndRemoveSmallest xs
163             val (d1,_,_) = x
164             val (d2,_,_) = sm
165         in
166           if d1>d2
167           then (sm,x::rest)
168           else (x,xs)
169         end
170       (* sortUList: (int * int * 'a) list -> (int * int * 'a) list
171        * Sorts the list according to relative deadlines
172        *)
173       fun sortUList xs =
174         let val (sm,rest) = findAndRemoveSmallest xs
175         in
176           if rest = []
177           then [sm]
178           else sm::sortUList rest
179         end
180       (* getExt: (_, _, 'a) -> 'a
181        * Gets the third element in a 3-tuple
182        *)
183       fun getExt(_,_,ext) = ext
184     in
185       (* mkPrio: (int * int * 'a) list -> 'a list
186        * Makes priorities for a given time step in the external
               representation ('a)
187        *)
188       fun mkPrio xs = map getExt (sortUList xs)
189     end
190     in
191     local
192       (* mkUListsAux: ((int * int * 'a) list * (int * int * 'a) list *
              int list * 'a list list)
193        *                                                    -> (
              int list * ''a list list)
```

```
194       * Auxilary function for 'mkULists' collecting time step sizes
              in 'As' and a list of
195       * priorities in 'prios'
196       *)
197      fun mkUListsAux(ds, ods, As, prios) =
198        if ds=ods
199        then (As, prios)
200        else mkUListsAux(timeStepList ds, ods, findSmallestD ds::As,
               mkPrio ds::prios)
201    in
202      (* mkULists: (int * int * 'a)list -> (int list * 'a list list)
203       * Function for creating urgency lists. It takes a list of
              elements (d,p,e) where
204       * d is the relative deadline, p is the period and e is the
              external representation
205       * for a given task
206       *)
207      fun mkULists ds =
208        let val (ts,ps) = mkUListsAux(timeStepList ds, ds, [
               findSmallestD ds], [mkPrio ds])
209        in  (rev ts, rev ps)
210        end
211    end
212  end
213
214  local
215    (* optULists: (int list * 'a list list) -> (int list * 'a list
          list)
216     * Function making optimized urgency lists, removing repeated
          elements
217     * from the priority list by adding their durrations
218     *)
219    fun optULists([t], [p]) = ([t],[p])
220      | optULists(t::ts, p::ps) =
221          let val (t1s, p1s) = optULists(ts,ps)
222          in
223            if hd p1s = p
224            then ((t+hd t1s)::tl t1s, p1s)
225            else (t::t1s, p::p1s)
226          end
227      | optULists _ = raise MixedLists
228    (* sumList: (int * int list) -> int list
229     * Function for adding a number to the head of an int list and
          adding
230     * the result of the addition to the rest of the elements in the
          list
231     * recursively
232     *)
233    fun sumList (_,[]) = []
234      | sumList (n,x::xs) = x+n::sumList(x+n,xs)
235  in
236    (* sumOptULists: (int * (int list * 'a list list)) -> (int list *
          'a list list)
237     * Function making the optimized urgency list and possibly adding
          a maximal offset
```

```
238     * to the time steps
239     *)
240    fun sumOptULists(MO,(ts,ps),opt) =
241      if(opt="O") then
242        let val (ots,ops) = optULists(ts,ps)
243        in (sumList(MO,ots), ops)
244        end
245      else
246        (sumList(MO,ts),ps)
247  end
248
249  (* FSMO: (int * int * int * 'a)list -> int
250   * Function for finding the smallest element in
251   * offsets or relative deadlines
252   *)
253  fun FSMO [] = raise EmptyList
254    | FSMO [(d,off,_,_)] = if off=0 then d else off
255    | FSMO((d,off,_,_)::rest) =
256        let val sm = FSMO rest
257        in
258          if off=0
259          then (if d<sm then d else sm)
260          else (if off<sm then off else sm)
261        end
262
263  (* offStep: (int * int * int * 'a * int) -> (int * int * int * 'a)
264   * Function taking a step during offset for a single element in
265   * a (distance,offset,period,ext) list
266   *)
267  fun offStep(d,off,per,ext,step) =
268        if off > 0
269        then
270          if off-step = 0
271          then (per, 0, per, ext)
272          else (d,off-step, per, ext)
273        else (((d-step-1)mod per)+ 1, off, per, ext)
274
275  (* offStep: (int * int * int * 'a)list -> (int * int * int * 'a)
       list
276   * Function taking a step during offset for a
277   * (distance,offset,period,ext) list
278   *)
279  fun offStepList lst = map (fn(d,off,p,e) => offStep(d,off,p,e,FSMO(
       lst))) lst
280
281  (* offZeros: (int * int * int * 'a)list -> bool
282   * Function checking if all offsets are zero
283   *)
284  fun offZeros [] = true
285    | offZeros((_,off,_,_)::rest) = off=0 andalso offZeros rest
286
287  (* findAndRemoveSmallest: (int * int * int * 'a)list -> (int * int
       * int list)
288   * Gives the element in the list with the smallest relative
       deadline or offset
```

```
289   * as well as the list without that element
290   *)
291  fun findAndRemoveSmallestOff [] = raise EmptyList
292    | findAndRemoveSmallestOff [e] = (e,[])
293    | findAndRemoveSmallestOff(x::xs) =
294    let val (sm,rest) = findAndRemoveSmallestOff xs
295        val (d1,o1,_,_) = x
296        val (d2,o2,_,_) = sm
297    in
298      if o1=0 andalso o2=0
299      then
300        if d1>d2
301        then (sm,x::rest)
302        else (x,xs)
303      else
304        if o1=0 andalso o2>0
305        then (x,xs)
306        else
307          if o1>0 andalso o2=0
308          then (sm,x::rest)
309          else
310            if o1>o2
311            then (sm,x::rest)
312            else (x,xs)
313    end
314  (* sortUListOff: (int * int * int * 'a) list -> (int * int * int * '
       a) list
315   * Sorts the list according to relative deadlines and offsets
316   *)
317  fun sortUListOff xs =
318    let val (sm,rest) = findAndRemoveSmallestOff xs
319    in
320      if rest = []
321      then [sm]
322      else sm::sortUListOff rest
323    end
324
325  (* getExtOff: (_, _, 'a) -> 'a
326   * Gets the fourth element in a 4-tuple
327   *)
328  fun getExtOff(_,_,_,ext) = ext
329
330  (* mkPrioOff: (int * int * int * 'a) list -> 'a list
331   * Makes priorities for a given time step during the offset in
332   * the external representation ('a)
333   *)
334  fun mkPrioOff xs = map getExtOff (sortUListOff xs)
335
336  (* mkOffUListsAux ((int * int * int * 'a) list * int list * 'a list
       list) ->
337   *    (int list * 'a list list)
338   *
339   * Auxilary function for 'mkOffULists' collecting time step sizes
       in 'As'
340   * and a list of priorities in 'prios' during offset
```

```
341
342  *)
343 fun mkOffULists Aux(lst,As,prios) =
344       if offZeros lst
345       then (rev As,rev prios,lst)
346       else mkOffULists Aux(offStepList lst, FSMO lst::As, mkPrioOff
              lst::prios)
347
348 (* mkOffULists: (int * int * 'a) list -> (int list * 'a list list)
349  * Function for creating offset urgency lists. It takes a list of
350  * elements (o,p,e) where
351  * o is the offset, p is the period and e is the external
        representation
352  * for a given task
353  *)
354 fun mkOffULists opeList = mkOffULists Aux(map (fn(off,p,e) => ((if
      off>0 then off else p),off,p,e)) opeList,[],[])
355
356 (* mkP: (int * int * int * 'a) -> (int * int * 'a)
357  * Function for making an element for the periodic list from and
       element
358  * of the offset list
359  *)
360 fun mkP (d,_,p,e) = (d,p,e)
361
362 (* mkPList: (int * int * int * 'a) list -> (int * int * 'a) list
363  * Function for making the periodic list from the offset list
364  *)
365 fun mkPList lst = map mkP lst
366
367 local
368   (* wh: ('a * int * 'a list) -> int
369    * Function finding a placement of an element in a list
370    *)
371   fun wh (i,_,[]) = raise Empty
372     | wh (i,n,x::xs) = if i=x then n else wh(i,n+1,xs)
373   (* whList: int list -> int list -> int list
374    * Function finding the placements of a list of elements
375    *)
376   fun whList [] lst = []
377     | whList (g::gs) lst = wh(g,1,lst)::whList gs lst
378   (* trans: int list -> int list list
379    * Function translating a list of priorities to their global ids
380    *)
381   fun trans gl pr = map (whList gl) pr
382   (* gl: 'a list -> 'a -> int
383    * Function finding the placement of a global id in a priority
         list
384    *)
385   fun gl gls s = wh(s,1,gls)
386 in
387   (* transPList: 'a list list * 'a list -> int list list
388    * Function translating a list of priorities in the external
389    * representation to a list of global ids
390    *)
```

```
391   fun transPList(pris,gls) = trans (map (gl gls) gls) (map (map (gl
          gls)) pris)
392 end
393
394 (* mkBothULists: (int * int * 'a) list ->
395  *   (int list * int list list) * (int list * int list list)
396  * Function for making the offset urgency list and the periodic
397  * urgency list from a list of 3-tuple elements (o,p,e), where o
398  * is the offset, p is the period and e is the external
399  * representation for each task
400  *)
401 fun mkBothULists opeList gls opt =
402   let val (ots,ops,lst) = mkOffULists opeList
403       val (optts, optps) = sumOptULists(0,(ots,ops),opt)
404       val lalie = if(List.length optts>0) then List.last optts else
              0
405       val plst = mkPList lst
406       val uplst = mkULists(plst)
407       val (perts,perps) = sumOptULists(lalie, uplst, opt)
408       val tpolist = transPList(optps,gls)
409       val tpper = transPList(perps,gls)
410   in ((optts,tpolist),(perts,tpper))
411   end
412
413 fun intStringList [] = raise EmptyList
414   | intStringList [h] = Int.toString(h)
415   | intStringList (h::t) = Int.toString(h)^","^intStringList t
416
417 local
418   fun strRepT [] = ""
419     | strRepT [t] = Int.toString t
420     | strRepT(t::ts) = Int.toString t ^ "," ^ strRepT ts
421   fun strRepP [] = ""
422     | strRepP [p] = strRepT p
423     | strRepP(p::ps) = strRepT p ^ "},{" ^ strRepP ps
424 in
425   fun strRep ((ots,ops),(pts,pps)) =
426       let val mostep = if(length ots>0) then List.nth(ots,((length
              ots)-1)) else 0
427           val offsteps = if(length ots>0) then strRepT ots else "0"
428           val offprios = if(length ots>0) then strRepP ops else
                  intStringList(hd pps)
429           val mxlstsz = if (length ots>length pts) then length ots
                  else length pts
430       in
431           "int[1,MN] pri[MN] = {"^intStringList(if(length ops>0)
                  then hd ops else hd pps)^"}; //EDF scheduling
                  priorities\n\nconst int NRSteps="^Int.toString(length
                   pts)^", NROffSteps="^(if(length ots>0) then Int.
                  toString(length ots) else "1")^", MAXOffStep="^Int.
                  toString(mostep)^", MAXStep="^Int.toString(List.nth(
                  pts,(length(pts)-1)))^", MAXListSize="^Int.toString(
                  mxlstsz)^";\n\nconst int[0,MAXOffStep] OffSteps[
                  NROffSteps] = {" ^ offsteps ^ "};\nconst int[1,MN]
                  OffPrios[NROffSteps][MN] = {{" ^ offprios ^ "}};\n\
```

```
                nconst int[0,MAXStep] Steps[NRSteps] = {" ^ strRepT
                pts ^ "};\nconst int[1,MN] Prios[NRSteps][MN] = {{" ^
                 strRepP pps ^ "}};\n\n"
432         end
433 end
434
435 fun mkGlList [] = []
436   | mkGlList(t::ts) = getTName t::mkGlList((ts))
437
438 fun mkOList([]) = []
439   | mkOList(t::ts) = (getO t, getPi t, getTName t)::mkOList((ts))
440
441 fun mkDynPri(app,opt) =
442   let val glist = mkGlList app
443       val olist = mkOList app
444       val blists = mkBothULists olist glist opt
445   in
446       strRep blists
447   end
448
449 fun mxExe ([],m) = m
450   | mxExe (t::rest,m) = if getWC t>m then mxExe(rest,getWC t) else
        mxExe(rest,m)
451
452 fun mxPi ([],m) = m
453   | mxPi (t::rest,m) = if getPi t>m then mxPi (rest,getPi t) else
        mxPi (rest,m)
454
455 fun mkDeclText((plat,app,mp,dp),opt) = "const int M = "^Int.
      toString(length plat)^";\nconst int N = "^Int.toString(maxTonP(
      plat,app,mp,dp))^";\nconst int MN = "^Int.toString(length app)^
      ";\n\nconst int[FP,EDF] processorScheduling[M] = {"^(mkProcSch
      plat)^"};\nconst int[0,MN] onPE[M][N] = "^mkOnPEText(plat,app,
      mp,dp)^"const int fpris[MN] = {"^mkFPris(1,app)^"};\nconst int
      pi[MN] = {"^mkPi app^"};\nconst int offset[MN] = {"^mkO app^"
      };\n\nconst bool origdep[MN][MN] = "^mkDepend(app,dp)^"\nbool
      depend[MN][MN] = "^mkDepend(app,dp)^"\n"^mkDynPri(app,opt)^"
      const int MaxExe="^Int.toString(mxExe(app,0))^";\nconst int
      MaxPi="^Int.toString(mxPi(app,0))^";\n\n"
456
457 fun extractTs [] = []
458   | extractTs(t::rest) = getTName t::extractTs(rest)
459
460 fun mkSysDeclP(_,[]) = ""
461   | mkSysDeclP(i,p::rest) = "Con"^Int.toString(i)^" = Control("^Int
        .toString(i)^");\nSyn"^Int.toString(i)^" = Synchronizer("^Int
        .toString(i)^");\nSch"^Int.toString(i)^" = Scheduler("^Int.
        toString(i)^");\n\n"^mkSysDeclP(i+1,rest)
462
463 fun mkSysDeclPInit(plat) = mkSysDeclP(1,plat)
464
465
466 fun mkSysDeclA(_,[],_,_) = "\n"
467   | mkSysDeclA(i,t::rest,(mp:mapping),plat) = getTName(t)^" = Task(
        "^Int.toString(getPIndex(1,(getPforT(getTName t,mp)),plat))^"
```

```
        ,␣"^Int.toString(i)^",␣"^Int.toString(getBC t)^",␣"^Int.
        toString(getWC t)^");\n"^mkSysDeclA(i+1,rest,mp,plat)
468
469   fun mkSysDeclAInit(app,mp,plat) = mkSysDeclA(1,app,mp,plat)
470
471   fun mkSysDeclEndA(_,[],sym) = ""
472     | mkSysDeclEndA(i,t::rest,sym) = getTName(t)^"␣"^sym^"␣"^
          mkSysDeclEndA(i+1,rest,sym)
473
474   fun mkSysDeclEndP(_,[],sym) = ""
475     | mkSysDeclEndP(i,p::rest,sym) = "Con"^Int.toString(i)^"␣"^sym^"␣
          "^"Syn"^Int.toString(i)^"␣"^sym^"␣"^"Sch"^Int.toString(i)^"␣"
          ^sym^"␣"^mkSysDeclEndP(i+1,rest,sym)
476
477   fun mkSysDeclEnd(app,plat,sym) = "system␣"^ mkSysDeclEndA(1,app,sym
      ) ^ mkSysDeclEndP(1,plat,sym) ^ "DynPri;\n"
478
479   fun mkFullSysDecl((plat,app,mp,dp),sym) = mkSysDeclPInit(plat) ^
      mkSysDeclAInit(app,mp,plat) ^ mkSysDeclEnd(app,plat,sym)
480
481   fun collectPre(is) = let val line = TextIO.inputLine is
482                        in if(line="//System-Dependent␣Decl\n") then "
                              " else line ^"\n"^collectPre is
483                        end
484
485   fun collectRealMid(is) = let val line = TextIO.inputLine is
486                            in if(line="//System-Dependent␣Inst\n")
                                  then "" else line ^collectRealMid is
487                            end
488
489   fun collectMid(is) = let val line = TextIO.inputLine is
490                        in if(line="//System-Independent␣Decl\n") then
                              collectRealMid(is) else collectMid(is)
491                        end
492
493   fun collectRealEnd(is) = let val line = TextIO.inputLine is
494                            in if(TextIO.endOfStream is) then line
                                  else line ^collectRealEnd is
495                            end
496
497   fun collectEnd(is) = let val line = TextIO.inputLine is
498                        in if(line="//System-Independent␣Inst\n") then
                              collectRealEnd(is) else collectEnd(is)
499                        end
500
501   fun readModel(filename) = let val is = TextIO.openIn filename
502                                 val pre = collectPre(is)
503                                 val middle = collectMid(is)
504                                 val ending = collectEnd(is)
505                             in (pre,middle,ending)
506                             end
507
508   fun collectFullModel(filename, S sys, opt, sym) =
509       let val (pre,middle,ending) = readModel filename
```

```
510     in pre ^ "//System-Dependent Decl\n" ^ mkDeclText(sys,opt) ^ "
            //System-Independent Decl\n" ^ middle ^ "//System-Dependent
            Inst\n" ^ mkFullSysDecl (sys,sym) ^ "//System-Independent
            Inst\n" ^ ending
511     end
512
513 fun mkFullModel(infile,sys,outfile,opt,sym) =
514     let val os = TextIO.openOut outfile
515     in (TextIO.output(os,collectFullModel(infile,sys,opt,sym)) ;
            TextIO.flushOut os)
516     end
517
518 fun getBusName(B(n,_,_)) = n
519
520 fun getBusSpeed(B(_,_,s)) = s
521
522 fun transBustoP(B(n,a,s)) = P(n,FP)
523
524 fun tMtoPplat(pes,bus) = transBustoP bus::pes
525
526 fun findCet (_,_,[]) = raise mappingError
527 | findCet(t,p,(tn,pn,bc,wc)::rest) = if t=tn andalso p=pn
528                                      then (bc,wc)
529                                      else findCet(t,p,rest)
530
531 fun findet (_,[],_) = raise noSuchT
532   | findet(n,(t,p)::rest,chr) = if n=t
533                                 then findCet(t,p,chr)
534                                 else findet(n,rest,chr)
535
536 fun tMtoPtask (MT(n,p,off),m,chrs) = let val (bc,wc) = findet(n,m,
        chrs)
537                                      in T(n,bc,wc,p,off)
538                                      end
539
540 fun findMT(_,[]) = raise noSuchT
541   | findMT(tn,MT(n,p,off)::rest) = if tn=n then MT(n,p,off) else
        findMT(tn,rest)
542
543 fun getPerforT(tn,mapp) = let val MT(_,p,_) = findMT(tn,mapp)
544                           in p
545                           end
546
547 fun getOfforT(tn,mapp) = let val MT(_,_,off) = findMT(tn,mapp)
548                          in off
549                          end
550
551 fun tMdep (_,[],_,_) = []
552   | tMdep (mapp,(t1,t2,d)::rest,m,(x,b)) = if d=0 orelse getPforT(
        t1,m)=getPforT(t2,m) then (tMdep(mapp,rest,m,(x,b))) else T(
        t1^"_"^t2,d div (getBusSpeed b),d div (getBusSpeed b),
        getPerforT(t1,mapp),getOfforT(t1,mapp))::tMdep(mapp,rest,m,(
        x,b))
553
554 fun tMmap ([],_,_) = []
```

```
555   | tMmap ((t1,t2,d)::rest,m,(x,b)) = if d=0 orelse getPforT(t1,m)=
        getPforT(t2,m) then tMmap(rest,m,(x,b)) else (t1^"_"^t2,
        getBusName b)::tMmap(rest,m,(x,b))
556
557 fun tMtoPapp ([],_,_) = []
558   | tMtoPapp (t::rest,m,chrs) = (tMtoPtask(t,m,chrs)::tMtoPapp(rest
        ,m,chrs))
559
560 fun tMtoPdep ([],_) = []
561   | tMtoPdep ((t1,t2,d)::rest,m) = if d=0 orelse getPforT(t1,m)=
        getPforT(t2,m) then (t1,t2)::tMtoPdep(rest,m) else (t1,t1^"_"
        ^t2)::(t1^"_"^t2,t2)::tMtoPdep(rest,m)
562
563 fun transMtoS(M(mpl,(mapp,mdep),m,chrs,prop)) =
564     let val p = tMtoPplat mpl
565         val a = tMtoPapp(mapp,m,chrs)@tMdep(mapp,mdep,m,mpl)
566         val d = tMtoPdep (mdep,m)
567         val ms = m @ tMmap(mdep,m,mpl)
568     in (S(p,a,ms,d),prop)
569     end
570
571 fun mkModelFromFile(infile,systemfile,outfile,opt,sym) =
572     let val os = TextIO.openOut outfile
573         val (psys,_) = transMtoS(parsef systemfile)
574         val cfm = collectFullModel(infile,psys,opt,sym)
575     in (TextIO.output(os,cfm) ; TextIO.flushOut os)
576     end
577
578 fun mkQuery(queryfile,systemfile) =
579     let val os = TextIO.openOut queryfile
580         val (_,prop) = transMtoS(parsef systemfile)
581     in
582         case prop of
583             Schedule => (TextIO.output(os,"A[]!missedDeadline");
                TextIO.flushOut os)
584           | Trace => (TextIO.output(os,"E<>missedDeadline"); TextIO
                .flushOut os)
585     end
586
587 fun printFullModel(infile,sys,opt,sym) = print (collectFullModel(
      infile,sys,opt,sym));
588
589
590 (* function handeling arguments *)
591 fun main() =
592 case CommandLine.arguments () of
593 (arg1::arg2::arg3::arg4::l) =>
594   let val pre = if ((size(arg3)>4) andalso substring(arg3,size(arg3
        )-4,4)=".xml") then substring(arg3,0,size(arg3)-4) else arg3
595   in
596   if(arg4="nt")
597     then (mkModelFromFile(arg2,arg1,pre^".xml","U","&lt;"); mkQuery
          (pre^".q",arg1))
598     else print "Usage: modelgen input_system system_template
          output_filename [nt]) \n\n"
```

```
599     end
600   |(arg1::arg2::arg3::l) =>
601     let val pre = if ((size(arg3)>4) andalso substring(arg3,size(arg3
          )-4,4)=".xml") then substring(arg3,0,size(arg3)-4) else arg3
602     in
603       (mkModelFromFile(arg2,arg1,pre^".xml","U",",");  mkQuery(pre^".q
            ",arg1))
604     end
605   | _ => print "Usage: modelgen input_system system_template
        output_filename [nt]) \n\n"
606
607   val _ = main();
```

B.3 Trace generator

Here is the abstract syntax used for trace generation, the lexer and parser definitions and the SML functions that make up the trace generator.

B.3.1 Abstract syntax

```
1  (* Absyn.sml: Abstract syntax for the MoVES trace generator
2     Aske Brekling 13/5/2008
3   *)
4
5  type tra = string*string*string
6  type transi = tra list
7  type cont = string*int
8  type status = (string*string) list
9  type state = status*(cont list)
10 type stt = state*transi
11 type trace = stt list
```

B.3.2 Lexer (lex)

```
1  {
2  (* MoVESlex.lex: lexer specification for MoVES trace generator
3     Aske Brekling 19/10/2009
4   *)
5
6  open Lexing tMoVESpar;
7
8  exception LexicalError of string * int * int (* (message, loc1,
       loc2) *)
9
10 fun lexerError lexbuf s =
11     raise LexicalError (s, getLexemeStart lexbuf, getLexemeEnd
           lexbuf);
12
13 }
14
15 rule Token = parse
16     [' ' '\t' '\n' '\r']  { Token lexbuf }
17   | "State:"              { STATE }
18   | "Transitions:"        { TRANS }
19   | "Delay:"              { DELAY }
20   | "{"                   { LBRACE }
21   | "}"                   { RBRACE }
22   | "("                   { LPAR }
23   | ")"                   { RPAR }
24   | "="                   { EQL }
25   | "->"                  { PREC }
26   | "-"                   { MINUS }
```

```
27    | "%"                       { PCT }
28    | "&"                       { AMP }
29    | "|"                       { PIPE }
30    | "."                       { DOT }
31
32    | ['0'-'9''-']['0'-'9']* { case Int.fromString (getLexeme lexbuf)
          of
33                                    NONE   => lexerError lexbuf "
                                          internal error"
34                                  | SOME i => INT i
35                              }
36    | ['a'-'z''A'-'Z''!'',''['']''<''*''+''_'':''>''?''#']['a'-'z''A
          '-'Z''0'-'9''_''['']''>'',':''+''!''?''|''&''#']*
37                              { NAME (getLexeme lexbuf) }
38    | eof                     { EOF   }
39    | _                       { lexerError lexbuf "Illegal symbol in
          input" }
40  and SkipToEndLine = parse
41      ['\n' '\r']             { () }
42     | (eof | '\^Z')          { () }
43     | _                      { SkipToEndLine lexbuf }
44  ;
```

B.3.3 Parser (yacc)

```
1   %{
2   (* tMoVESpar.grm: parser specification for MoVES trace generator
3      Aske Brekling 19/10/2009
4    *)
5
6     open tAbsyn;
7   %}
8
9   %token <int> INT                          // Accepting numbers
10  %token <string> NAME                      // Accepting names
11  %token STATE TRANS DELAY                  // Tokens for system
        keywords
12  %token LBRACE RBRACE LPAR RPAR            // Tokens for application
        keywords
13  %token EQL MINUS PREC PCT AMP PIPE DOT    // Tokens for application
        keywords
14  %token EOF                                // Token for end of file
15
16
17  %start Main
18
19  %type <tAbsyn.trace> Main stlist
20  %type <tAbsyn.stt> st
21  %type <tAbsyn.state> stat
22  %type <tAbsyn.status> status
23  %type <tAbsyn.transi> transitions
24  %type <tAbsyn.tra> trans
25  %type <string> extras
```

```
26 %type <string> extra
27 %type <string> cextras
28 %type <string> cextra
29 %type <tAbsyn.cont list> content
30 %%
31
32 // The main program
33 Main:
34     stlist EOF                          { $1 }
35   | EOF                                 { [] }
36 ;
37
38 stlist:
39   st            { ([$1]) }
40   | st stlist   { $1::$2 }
41 ;
42
43 st:
44     STATE stat                          { ($2,[]) }
45   | STATE stat TRANS transitions        { ($2,$4) }
46   | STATE stat DELAY INT                { ($2,[]) }
47   | STATE stat DELAY INT DOT INT        { ($2,[]) }
48 ;
49
50 stat:
51   LPAR status RPAR content      { ($2,$4) }
52 ;
53
54 status:
55    /* empty */          { [] }
56  | NAME DOT NAME status { ($1,$3)::$4 }
57 ;
58
59 content:
60         { [] }
61   | cextras EQL INT content             { ($1,$3)::$4 }
62   | cextras EQL INT DOT INT content     { ($1,$3)::$6 }
63   | cextras EQL MINUS INT content       { ($1,$4)::$5 }
64 ;
65
66 cextras:
67         { "" }
68   | cextra cextras      { $1^$2 }
69 ;
70
71 cextra:
72     NAME        { $1 }
73   | DOT         { "." }
74 ;
75
76 transitions:
77     trans                       { [$1] }
78   | trans transitions   { $1::$2 }
79 ;
80
```

```
81  trans:
82    cextras PREC cextras LBRACE extras RBRACE  { ($1,$3,$5) }
83  ;
84
85  extras:
86          { "" }
87    | extra extras                { $1^$2 }
88  ;
89
90  extra:
91      NAME         { $1 }
92    | INT          { Int.toString($1) }
93    | LPAR         { "(" }
94    | RPAR         { ")" }
95    | EQL          { "=" }
96    | MINUS        { "-" }
97    | PCT          { "%" }
98    | AMP          { "&" }
99    | PIPE         { "|" }
100   | DOT          { "." }
101 ;
```

B.3.4 Auxiliary functions for the lexer/parser of the trace generator

```
1   (* Lexer and parser for the MoVES trace generator using mosmllex
        and mosmlyac
2      Aske Brekling 13/5/2008
3    *)
4   open tAbsyn;
5
6   (* Plain parsing from a string, with poor error reporting *)
7
8   fun parse str =
9       let val lexbuf = Lexing.createLexerString str
10          val expr   = tMoVESpar.Main tMoVESlex.Token lexbuf
11      in
12          Parsing.clearParser();
13          expr
14      end
15      handle exn => (Parsing.clearParser(); raise exn);
16
17
18  (* Fancy parsing from a file; show the offending program piece on
        error *)
19
20  fun parseExprReport file stream lexbuf =
21      let val expr =
22              tMoVESpar.Main tMoVESlex.Token lexbuf
23              handle
24                 Parsing.ParseError f =>
25                     let val pos1 = Lexing.getLexemeStart lexbuf
26                         val pos2 = Lexing.getLexemeEnd lexbuf
```

```
27                      in
28                          Location.errMsg (file, stream, lexbuf)
29                                          (Location.Loc(pos1, pos2))
30                                          "Syntax error."
31                      end
32                | tMoVESlex.LexicalError(msg, pos1, pos2) =>
33                      if pos1 >= 0 andalso pos2 >= 0 then
34                          Location.errMsg (file, stream, lexbuf)
35                                          (Location.Loc(pos1, pos2))
36                                          ("Lexical error: " ^ msg)
37                      else
38                          (Location.errPrompt ("Lexical error: " ^ msg
                                ^ "\n\n");
39                           raise Fail "Lexical error");
40      in
41          Parsing.clearParser();
42          expr
43      end
44      handle exn => (Parsing.clearParser(); raise exn);
45
46  (* Parse a program from a string, with error reporting *)
47
48  fun parses str =
49      parseExprReport "" (BasicIO.std_in) (Lexing.createLexerString
            str);
50
51  (* Create lexer from instream *)
52
53  fun createLexerStream (is : BasicIO.instream) =
54      Lexing.createLexer (fn buff => fn n => Nonstdio.buff_input is
            buff 0 n)
55
56  (* Parse a program from a file, with error reporting *)
57
58  fun parsef file =
59      let val is     = Nonstdio.open_in_bin file
60          val expr   = parseExprReport file is (createLexerStream is)
61                       handle exn => (BasicIO.close_in is; raise exn)
62      in
63          BasicIO.close_in is;
64          expr
65      end;
```

B.3.5 MoVES trace generator

Interesting signature (based on types from the abstract syntax):

```
mkTrace: trace -> Unit
(* Prints the MoVES trace to the user with the built-in function
   print. *)
```

```
1  open tparser;
2  exception mixedLists
3  exception noSuchT
4
5  fun getFirst(s,t) = s
6  fun getSecond(s,t) = t
7
8  fun getStates (s:trace) = map getFirst s
9
10 fun filterT [] = []
11   | filterT((s,t)::rest) = if size(s)<4 then (s,t)::filterT rest
       else (if substring(s,0,3)="Con" orelse substring(s,0,3)="Syn"
        orelse substring(s,0,3)="Sch" then filterT rest else (if
       size(s) < 6 then (s,t)::filterT rest else (if substring(s
       ,0,6)="DynPri" then filterT rest else (s,t)::filterT rest)))
12
13 fun getSt (s:trace) = map getFirst (getStates s)
14 fun getTSt (s:trace) = map filterT (getSt s)
15
16 fun getCont (s:trace) = map getSecond (getStates s)
17
18 fun exTM [] = 0
19   | exTM((s,i)::rest) = if s="TM" then i else exTM rest
20
21 fun exTMs c = map exTM c
22
23 fun getLastSame ([a],b::rest) = [(a,b)]
24   | getLastSame (a1::a2::arest,b1::b2::brest) = if a1=a2 then
       getLastSame(a2::arest,b2::brest) else (a1,b1)::getLastSame(a2
       ::arest,b2::brest)
25   | getLastSame _ = raise mixedLists
26
27 fun maxS(i,[]) = i
28   | maxS (i,(t,_)::rest) = if (size t>i) then maxS(size t,rest)
       else maxS(i,rest)
29
30 fun getMaxT (s:trace) = List.last(exTMs(getCont s))
31 fun getMaxS (s:trace) = maxS(0,hd(getTSt s))
32
33 fun space 0 = ""
34   | space n = " "^space(n-1)
35
36 fun times(i,j) = if i=j then Int.toString(i mod 10) else
37                  (if i mod 2=0 then Int.toString(i mod 10) else " "
                     )^times(i+1,j)
38
39 fun mkTime (s:trace) = space(getMaxS s)^" | "^times(0,getMaxT s)
40
41 fun getSforT "Done" = " "
42   | getSforT "DoneU" = " "
43   | getSforT "Running" = "+"
44   | getSforT "RunningU" = "+"
45   | getSforT "RunningA" = "+"
46   | getSforT "Released" = "O"
47   | getSforT "Offset" = " "
```

```
48    | getSforT "OffsetU" = "␣"
49    | getSforT "Dmiss" = "X"
50    | getSforT _ = "P"
51
52  fun findTforT(_,[]) = raise noSuchT
53    | findTforT(ts,(t,s)::rest) = if ts=t then (getSforT s) else
          findTforT(ts,rest)
54
55  fun tmln(_,_,_,[]) = ""
56    | tmln(n,sym,ts,(i,st)::rest) = if n=i then findTforT(ts,st)^tmln
          (n+1,findTforT(ts,st),ts,rest) else sym^tmln(n+1,sym,ts,(i,st
          )::rest)
57
58  fun mkTimeT(ts,s:trace) = ts ^ space(getMaxS s-size ts)^ "␣|␣"^tmln
        (0,"␣",ts,getLastSame(exTMs(getCont s),getTSt s))
59
60  fun mkTimeTs([],_) = ""
61    | mkTimeTs(t::ts,s) = mkTimeT(t,s)^"\n"^mkTimeTs(ts,s)
62
63
64  fun getTfromS(s:trace) = map (fn(a,b)=>a) (hd(getTSt s))
65
66  fun mkTrace [] = ""
67    | mkTrace (s:trace) = mkTime(s)^"\n"^mkTimeTs(getTfromS s,s)
68
69
70
71  (* function handelling arguments *)
72  fun main() =
73  case CommandLine.arguments () of
74  (arg1::l) =>
75    print(mkTrace(parsef arg1))
76  | _ => print "Usage:␣tracegen␣trace-file␣\n\n"
77
78  val _ = main();
```

Appendix C

Batch Scripts for MoVES

This appendix gives the batch scripts used to invoke the different parts of the MoVES framework. There is one for Windows users and another for Linux users.

C.1 For Windows Users

```
1  @echo off
2  set NrArgs=0
3  for %%a in (%*) do set /a NrArgs+=1
4  if /i %NrArgs% == 0 (
5   ECHO Not enough arguments! type 'moves -help' for instructions
6   GoTo :End_Of_Batch
7  )
8  if /i %1 == -help (
9    echo usage: moves [ verification_options ] system-file
10   echo  verification_options:
11   echo   -sw 'stop watch automata verification'
12   echo   -drt 'discretization of running time verification'
13   echo   -nc 'no clocks verification'
14   echo.
15   echo  each of these options can be followed by -nt for no trace
         generation
16   echo.
17   echo  examples:
```

```
18    echo ------ Default Options - 'no clocks' ------
19    echo   moves a
20    echo    system-file: a
21    echo    v-structure: testTemplate.xml
22    echo.
23    echo ------ Stop-Watch Options ------
24    echo   moves -sw a
25    echo    system-file: a
26    echo    v-structure: testTemplateSW.xml
27    echo    verification engine: verifyta-sw
28    echo.
29    echo ------ Discretization Options ------
30    echo   moves -drt a
31    echo    system-file: a
32    echo    v-structure: testTemplateDRT.xml
33    echo    verification engine: verifyta
34    echo.
35    echo ------ No-Clocks Options 'default' ------
36    echo   moves -nc a
37    echo    system-file: a
38    echo    v-structure: testTemplateDRT.xml
39    echo    verification engine: verifyta
40    echo.
41    echo ------ Stop-Watch and no-trace Options ------
42    echo   moves -sw-nt a
43    echo    system-file: a
44    echo    v-structure: testTemplateSW.xml
45    echo    verification engine: verifyta-sw
46    echo   *no trace will be generated
47    echo.
48    echo ------ Discretization and no-trace Options ------
49    echo   moves -drt-nt a
50    echo    system-file: a
51    echo    v-structure: testTemplateDRT.xml
52    echo    verification engine: verifyta
53    echo   *no trace will be generated
54    echo.
55    echo ------ No-Clocks and no-trace Options ------
56    echo   moves -nc-nt a
57    echo    system-file: a
58    echo    v-structure: testTemplateDRT.xml
59    echo    verification engine: verifyta
60    echo   *no trace will be generated
61    GoTo :End_Of_Batch
62  )
63  if /i %NrArgs% == 1 if exist "%1" (
64     modelgen %1 testTemplate.xml %1
65     verifyta %1.xml %1.q -qst 1 2> %1.trace
66     tracegen %1.trace
67     GoTo :End_Of_Batch
68  )
69  if /i %NrArgs% == 1 if not exist "%1" (
70    echo File: '%1' does not exist
71    Goto :End_Of_Batch
72  )
```

```
73  if /i %NrArgs% GTR 2 (
74     echo Too many arguments! type 'moves -help' for instructions
75     GoTo :End_Of_Batch
76  )
77  if /i %1 == -sw  if exist "%2" (
78       modelgen %2 testTemplateSW.xml %2
79       verifyta -sw %2.xml %2.q -qst 1 2> %2.trace
80       tracegen %2.trace
81       GoTo :End_Of_Batch
82
83  )
84  if /i %1 == -drt  if exist "%2" (
85       modelgen %2 testTemplateDRT.xml %2
86       verifyta %2.xml %2.q -qst 1 2> %2.trace
87       tracegen %2.trace
88       GoTo :End_Of_Batch
89  )
90  if /i %1 == -nc if exist "%2" (
91       modelgen %2 testTemplate.xml %2
92       verifyta %2.xml %2.q -qst 1 2> %2.trace
93       tracegen %2.trace
94       GoTo :End_Of_Batch
95  )
96  if /i %1 == -sw-nt  if exist "%2" (
97       modelgen %2 testTemplateSW.xml %2 nt
98       verifyta -sw %2.xml %2.q -qs
99       GoTo :End_Of_Batch
100
101 )
102 if /i %1 == -drt-nt  if exist "%2" (
103      modelgen %2 testTemplateDRT.xml %2 nt
104      verifyta %2.xml %2.q -qs
105      GoTo :End_Of_Batch
106 )
107 if /i %1 == -nc-nt if exist "%2" (
108      modelgen %2 testTemplate.xml %2 nt
109      verifyta %2.xml %2.q -qs
110      GoTo :End_Of_Batch
111 )
112 if /i %NrArgs% == 2 if not exist "%2" (
113   echo File: '%2' does not exist
114   Goto :End_Of_Batch
115 )
116 echo Option: %1 not recognized
117
118 :End_Of_Batch
```

C.2 For Linux Users

```
1  #!/bin/sh
2
3  if [ $# -eq 1 ];
4  then
```

```
5     if [ $1 = "-help" ]
6     then
7       echo -e "usage: moves [ verification_options ] file-prefix \
            n verification_options: \n  -sw 'stop watch automata
            verification '\n  -drt 'discretization of running time
            verification '\n  -nc 'no clocks verification '\n\n examples
            :\n ------ Default Options - 'no clocks ' ------\n  moves a\n
            file-prefix: a  \n  v-structure: testTemplate.xml\n\n
            ------ Stop-Watch Options ------\n  moves -sw a\n  file-
            prefix: a\n  v-structure: testTemplateSW.xml\n
            verification engine: verifyta -sw \n\n ------ Discretization
            Options ------\n  moves -drt a\n  file-prefix: a\n  v-
            structure: testTemplateDRT.xml\n  verification engine:
            verifyta \n\n ------ No-Clocks Options 'default ' ------\n
            moves -nc a\n  file-prefix: a\n  v-structure:
            testTemplateDRT.xml\n  verification engine: verifyta \n\n
            ------ Stop-Watch and no-trace Options ------\n  moves -sw-nt
            a\n  file-prefix: a\n  v-structure: testTemplateSW.xml\n
            verification engine: verifyta -sw\n  *no trace will be
            generated \n\n ------ Discretization and no-trace Options
            ------\n  moves -drt-nt a\n  file-prefix: a\n  v-structure
            : testTemplateDRT.xml\n  verification engine: verifyta \n
            *no trace will be generated \n\n ------ No-Clocks and no-
            trace Options ------\n  moves -nc-nt a\n  file-prefix: a\n
            v-structure: testTemplateDRT.xml\n  verification engine:
            verifyta \n  *no trace will be generated "
8     else
9       if [ -f $1 ];
10      then
11        modelgen $1 testTemplate.xml $1
12        verifyta $1.xml $1.q -qst 1 2> $1.trace
13        tracegen $1.trace
14      else
15        echo "File: " $1 " does not exist"
16      fi
17    fi
18  else
19    if [ $# -eq 2 ];
20    then
21      if [ $1 = "-sw" ];
22      then
23        if [ -f $2 ];
24        then
25          modelgen $2 testTemplateSW.xml $2
26          verifyta -sw $2.xml $2.q -qst 1 2> $2.trace
27          tracegen $2.trace
28        else
29          echo "File: " $2 " does not exist"
30        fi
31      else
32        if [ $1 = "-drt" ];
33        then
34          if [ -f $2 ];
35          then
36            modelgen $2 testTemplateDRT.xml $2
```

```
37        verifyta $2.xml $2.q -qst 1 2> $2.trace
38        tracegen $2.trace
39       else
40         echo "File: " $2 " does not exist"
41       fi
42      else
43       if [ $1 = "-nc" ];
44       then
45        if [ -f $2 ];
46        then
47         modelgen $2 testTemplate.xml $2
48         verifyta $2.xml $2.q -qst 1 2> $2.trace
49         tracegen $2.trace
50        else
51         echo "File: " $2 " does not exist"
52        fi
53       else
54        if [ $1 = "-nc-nt" ];
55        then
56         if [ -f $2 ];
57         then
58          modelgen $2 testTemplate.xml $2 nt
59          verifyta $2.xml $2.q -qs
60         else
61          echo "File: " $2 " does not exist"
62         fi
63        else
64         if [ $1 = "-sw-nt" ];
65         then
66          if [ -f $2 ];
67          then
68           modelgen $2 testTemplateSW.xml $2
69           verifyta-sw $2.xml $2.q -qs
70          else
71           echo "File: " $2 " does not exist"
72          fi
73         else
74          if [ $1 = "-drt-nt" ];
75          then
76           if [ -f $2 ];
77           then
78            modelgen $2 testTemplateDRT.xml $2 nt
79            verifyta $2.xml $2.q -qs
80           else
81            echo "File: " $2 " does not exist"
82           fi
83          else
84           echo "Option: "$1" not recognized"
85          fi
86         fi
87        fi
88       fi
89      fi
90     fi
91   else
```

```
92     echo "Wrong number of arguments! type 'moves --help' for
           instructions"
93   fi
94 fi
```

Bibliography

[1] Yasmina Abdeddaïm, Eugene Asarin, and Oded Maler. Scheduling with Timed Automata. *Theoretical Computer Science*, 354(2):272–300, 2006.

[2] Yasmina Abdeddaïm and Oded Maler. Job-Shop Scheduling Using Timed Automata. In *Procedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102, pages 478–492. Springer Berlin / Heidelberg, 2001.

[3] Karine Altisen and Stavris Tripakis. Implementation of Timed Automata: An Issue of Semantics or Modeling? In Paul Pettersson and Wang Yi, editors, *Formal Modeling and Analysis of Timed Systems*, volume 3829 of *Lecture Notes in Computer Science*, pages 273–288. Springer Berlin / Heidelberg, 2005.

[4] Rajeev Alur, Costas A. Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The Algorithmic Analysis of Hybrid Systems. *Theoretical Computer Science*, 138:3–34, 1995.

[5] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[6] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. TIMES - A Tool for Modelling and Implementation of Embedded Systems. *Lecture Notes in Computer Science*, 2280:460–464, 2002.

[7] Neil C. Audsley, Alan Burns, Robert I. Davis, Ken W. Tindell, and Andy J. Wellings. Fixed Priority Pre-emptive Scheduling: An Historical Perspective. *Real-Time Syst.*, 8(2-3):173–198, 1995.

[8] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on Uppaal. *Lecture Notes in Computer Science*, 3185:200–236, 2004.

[9] Aske Brekling, Michael R. Hansen, and Jan Madsen. Models and Formal Verification of Multiprocessor System-on-Chips. *Journal of Logic and Algebraic Programming*, 77(1-2):1 – 19, 2008.

[10] Aske W. Brekling, Michael R. Hansen, and Jan Madsen. A Timed-Automata Semantics for a System-Level MPSoC Model. In *proceedings of the 18th Nordic Workshop on Programming Theory (NWPT 2006)*, 2006.

[11] Aske W. Brekling, Michael R. Hansen, and Jan Madsen. Analysis of Quantitative Properties of Hardware Specifications. In *proceedings of the 21st Nordic Workshop on Programming Theory (NWPT 2009)*, pages 92–95, 2009.

[12] Aske W. Brekling, Michael R. Hansen, and Jan Madsen. MoVES - A Framework for Modelling and Verifying Embedded Systems. In *2009 International Conference on Microelectronics*, pages 143–146. IEEE Computer Society, 2009.

[13] Alan Burns and Andy J. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley Longman, Redwood City, CA, USA, 3 edition, 2001.

[14] Alan Burns and Andy J. Wellings. Delivering Real-Time Behaviour. In *Domain Modeling and the Duration Calculus*, pages 1–50, 2007.

[15] Anton Cervin and Karl-Erik Årzén. TrueTime: Simulation tool for performance analysis of real-time embedded systems. In Gabriela Nicolescu and Pieter J. Mosterman, editors, *Model-Based Design for Embedded Systems*. CRC Press, November 2009.

[16] Computers and Real-Time Group University of Cantabria (Spain). MAST - Modeling and Analysis Suite for Real-Time Applications. Project website. `http://mast.unican.es`, 2008.

[17] Rene L. Cruz. A Calculus for Network Delay. *IEEE Transactions on Information Theory*, 37(1):114–141, 1991.

[18] Raymond A. Cunninghame-Greene. Minimax Algebra. *Lecture Notes in Economics and Mathematical Systems*, 166, 1979.

[19] Alexandre David, Jacob Illum, Kim G. Larsen, and Arne Skou. Model-Based Framework for Schedulability Analysis Using Uppaal 4.1. In Gabriela Nicolescu and Pieter J. Mosterman, editors, *Model-Based Design for Embedded Systems*, pages 93–119. CRC Press, 2010.

[20] Conrado Daws, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos.
`http://www-verimag.imag.fr/DIST-TOOLS/TEMPO/kronos/`, 2002.

[21] Design and Sweden Analysis of Real-Time Systems, Uppsala University. Times - A Tool for Modeling and Implementation of Embedded Systems. Project website. `http://www.timestool.com/`, 2007.

[22] Elena Fersman, Pavel Krcal, Paul Pettersson, and Wang Yi. Task Automata: Schedulability, Decidability and Undecidability. *Information and Computation*, 205(8):1149–1172, 2007.

[23] Elena Fersman, Paul Pettersson, and Wang Yi. Timed Automata with Asynchronous Processes: Schedulability and Decidability. *Lecture Notes in Computer Science*, 2280:67–82, 2002.

[24] J. Carlos Palencia Guitérrez, J. Javier Gutiérrez García, and Michael González Harbour. On the Schedulability Analysis for Distributed Hard Real-Time Systems. In *Proceedings of the 9th Euromicro Workshop on Real-Time Systems, Toledo, Spain*, pages 136–143, 1997.

[25] J. Carlos Palencia Guitérrez and Michael González Harbour. Schedulability Analysis for Tasks with Static and Dynamic Offsets. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1998.

[26] Lone Halkjaer, Karen Haervi, and Anna Ingolfsdottir. Verification of the LegOS Scheduler using UPPAAL. *Electronic Notes in Theoretical Computer Science*, 39(3):273–292, 2000.

[27] Michael R. Hansen, Jan Madsen, and Aske Brekling. Semantics and Verification of a Language for Modelling Hardware Architectures. In Cliff Jones, Zhiming Liu, and Jim Woodcock, editors, *Formal Methods and Hybrid Real-Time Systems*, volume 4700 of *Lecture Notes in Computer Science*, pages 300–319. Springer Berlin / Heidelberg, 2007.

[28] Michael R. Hansen and Hans Rischel. *Introduction to Programming using SML*. Addison Wesley Longman, 1999.

[29] Michael González Harbour, J. Javier Gutiérrez García, J. Carlos Palencia Guitérrez, and José Maria Drake Moyano. MAST: Modeling and Analysis Suite for Real Time Applications. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pages 125–134, 2001.

[30] Christian Haubelt, Joachim Falk, Joachim Keinert, Thomas Schlichter, Martin Streubühr, Andreas Deyhle, Andreas Hadert, and Jürgen Teich. A SystemC-Based Design Methodology for Digital Signal Processing Systems. *EURASIP Journal on Embedded Systems*, 2007(1):15–15, 2007.

[31] Martijn Hendriks and Marcel Verhoef. Timed Automata Based Analysis of Embedded System Architectures. In *Procedings of the 20th International Parallel and Distributed Processing Symposium*, page 8 pp. IEEE Computer Society, 2006.

[32] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. System Level Performance Analysis - the SymTA/S Approach. In *Proceedings of IEE Computers and Digital Techniques*, volume 152, pages 148–166. IEE, 2005.

[33] Fabiano Hessel, Vitor M. da Rosa, Igor M. Reis, Ricardo Planner, Cesar A. M. Marcon, and Altamiro A. Susin. Abstract RTOS Modeling for Embedded Systems. In *Proceedings of the 15th IEEE International Workshop on Rapid System Prototyping (RSP'04)*, pages 210–216. IEEE Computer Society, 2004.

[34] C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall International Series in Computer Science, 1998.

[35] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[36] Ieee. *Std 1076-2000: IEEE Standard VHDL Language Reference Manual*. IEEE, 2000.

[37] AVACS H1/2 iSAT Developer Team. iSAT... Tight Integration of Satisfiability & Constraint Solving. Project website. `http://isat.gforge.avacs.org/`, 2010.

[38] Torsten Kempf, Malte Doerper, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Tim Kogel, and Bart Vanthournout. A Modular Simulation Framework for Spatial and Temporal Task Mapping onto Multi-Processor SoC Platforms. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'05)*, volume 2, pages 876–881. IEEE Computer Society, 2005.

[39] Pavel Krcál and Wang Yi. Decidable and Undecidable Problems in Schedulability Analysis Using Timed Automata. In *In proceedings of 10th International Conference, TACAS'04 LNCS*, volume 2988, pages 236–250. Springer Berlin / Heidelberg, 2004.

[40] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[41] Thorsten Grötkerand Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[42] Chang L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, 1973.

[43] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[44] Mirko Loghi, Federico Angiolini, Davide Bertozzi, Luca Benini, and Roberto Zafalon. Analyzing On-Chip Communication in a MPSoC Environment. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'04)*, pages 752–757. IEEE Computer Society, 2004.

[45] Jan Madsen, Michael R. Hansen, and Aske W. Brekling. A Modelling and Analysis Framework for Embedded Systems. In Gabriela Nicolescu and Pieter J. Mosterman, editors, *Model-Based Design for Embedded Systems*, pages 121–143. CRC Press, 2010.

[46] Jan Madsen, Michael R. Hansen, Kristian S. Knudsen, Jens E. Nielsen, and Aske W. Brekling. System-level Verification of Multi-Core Embedded Systems using Timed-Automata. In *Proceedings of the 17th International Federation of Automatic Control World Congress (IFAC'08)*, 2008.

[47] Jan Madsen, Shankar Mahadevan, and Kashif Virk. Network-Centric System-Level Model for Multiprocessor SoC Simulation. In *Interconnect-Centric Design for Advanced SoC and NoC*, pages 341–365. Kluwer Academic, 2004.

[48] Jan Madsen, Kashif Virk, and Mercury J. Gonzalez. A SystemC-Based Abstract Real-Time Operating System Model for Multiprocessor System-on-Chip. In *Multiprocessor System-on-Chip*, pages 283–312. Morgan Kaufmann, 2004.

[49] Shankar Mahadevan, Michael Storgaard, Jan Madsen, and Kashif Virk. ARTS: A System-Level Framework for Modeling MPSoC Components and Analysis of their Causality. In *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'05)*, pages 480–483. IEEE Computer Society, 2005.

[50] Shankar Mahadevan, Kashif Virk, and Jan Madsen. ARTS: A SystemC-based Framework for Multiprocessor Systems-on-Chip Modelling. *Design Automation for Embedded Systems*, 11(4):285–311, 2007.

[51] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[52] R. Le Moigne, Oliver Pasquier, and Jean Paul Calvez. A Generic RTOS Model for Real-Time Systems Simulation with SystemC. In *Proceedings*

of the Conference on Design, Automation and Test in Europe (DATE'04), volume 3, pages 82–87. IEEE Computer Society, 2004.

[53] Gabriela Nicolescu and Pieter J. Mosterman. *Model-Based Design for Embedded Systems*. CRC Press, 2010.

[54] Tolga Ovatman, Aske W. Brekling, and Michael R. Hansen. Analysis of Costs of Embedded Systems: Experiments with Priced Timed Automata. In *Formal Foundations of Embedded Software and Component-Based Software Architectures, FESCA@ETAPS*, pages 1–14, 2008.

[55] Denmark Peter Sestoft, IT University of Copenhagen. Moscow ML. Project website. `http://www.itu.dk/people/sestoft/mosml.html`, 1995.

[56] Andy D. Pimentel, Cagkan Erbas, and Simon Polstra. A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels. *IEEE Transactions on Computers*, 55(2):99–112, 2006.

[57] Andy D. Pimentel, Louis O. Hertzberger, Paul Lieverse, Pieter van derWolf, and Ed F. Deprettere. Exploring Embedded-Systems Architectures with Artemis. *IEEE Computer*, 34(11):57–63, 2001.

[58] Paul Pop, Petru Eles, and Zebo Peng. Schedulability-Driven Communication Synthesis for Time Triggered Embedded Systems. *Real-Time Systems*, 26:297–325, 2004.

[59] Patrick Schaumont and Ingrid Verbauwhede. Domain Specific Tools and Methods for Application in Security Processor Design. *Design Automation for Embedded Systems 7*, pages 365–383, 2002.

[60] Marcus T. Schmitz, Bashir M. Al-Hashimi, and Petru Eles. *System-Level Design Techniques for Energy-Efficient Embedded Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[61] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-Time Calculus for Scheduling Hard Real-Time Systems. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS 2000)*, volume 4, pages 101–104, Geneva, Switzerland, 2000.

[62] Donald E. Thomas and Philip R. Moorby. *The VERILOG Hardware Description Language*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[63] Ken Tindell. Adding Time-Offsets to Schedulability Analysis. *Technical Report YCS 221*, 1994.

[64] Ken Tindell, Alan Burns, and Andy J. Wellings. An Extendible Approach for Analysing Fixed Priority Hard-Real-Time tasks. *Real-Time Systems*, 6(2):133–151, 1994.

[65] Ken Tindell and John Clark. Holistic Schedulability Analysis for Hard Real-Time Systems. *Microprocessing and Microprogramming*, 40(2-3):117–134, 1994.

[66] Aalborg University and Uppsala University. UPPAAL. `http://www.uppaal.com`, 2009.

[67] Ernesto Wandeler and Lothar Thiele. Real-Time Calculus (RTC) Toolbox. `http://www.mpa.ethz.ch/Rtctoolbox`, 2006.

[68] Sergio Yovine. Kronos: A Verification Tool for Real-Time Systems. *International Journal of Software Tools for Technology Transfer*, 1(1/2):123–133, 1997.